ARBEITSPAPIERE DER NORDAKADEMIE

ISSN 1860-0360

Nr. 2012-04

SStDSL, eine DSL für Schnittstellen mit PetitParser

Hartmut Krasemann
Johannes Brauer
Christoph Crasemann

März 2012

Eine elektronische Version dieses Arbeitspapiers ist verfügbar unter
http://www.nordakademie.de/arbeitspapier.html

NORDAKADEMIE
HOCHSCHULE DER WIRTSCHAFT
THEORIE PRAXIS

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

SStDSL, eine DSL für Schnittstellen mit Petit-Parser

Hartmut Krasemann, Johannes Brauer, Christoph Crasemann

12. März 2012

Inhaltsverzeichnis

1 Einleitung **2**

2 Das SStDSL-Referenzmodell für Schnittstellen **3**
2.1 Nachrichten-Typen 5
2.2 Nachrichten-Router 5
2.3 Synchronizität 6
2.4 Publish/Subscribe 7
2.5 Minimalistik: nur eine Empfangsqueue 7
2.6 Versionierung 7
2.7 Spezifikation der Schnittstelle (Interface) 8
2.8 Querschnittliche Information 9
2.9 Zustellgarantie und Dauerhaftigkeit 9
2.10 Transaktionalität 10
2.11 Über die Rolle der Commands 10

3 Beispiel: Die Statechart-Schnittstelle des MicrowaveOven **10**
3.1 Der MicrovaweOven der AuDSL 10
3.2 Die INTERFACE Spezifikation 11
3.3 Aufrufschnittstelle 13

4 Die SStDSL **14**
4.1 Der Parser der SStDSL 15
4.2 Das semantische Modell 16
4.3 Fehlerbehandlung 18
4.4 Logging und Zeitmessung 19

5 Erweiterungen der SStDSL **19**

6 Erfahrungen **20**

7 Ausblick **21**

8 Danksagung **22**

Abstract

Schnittstellen zwischen Anwendungen werden heute in der Praxis meist auf Basis sehr fundamentaler Techniken programmiert. Das führt zu einer großen Menge technischen Codes im Anwendungsprogramm. Wir konstruieren deshalb eine eingebettete Konfigurations-DSL für Schnittstellen (die SStDSL) mit dem Ziel, die Anwendung von dem technischen Schnittstellencode zu entlasten.

Die Implementierung mit Hilfe eines Parserkombinators erlaubt uns, ohne weiteres eine angemessene neue Syntax für die Schnittstellenbeschreibung innerhalb des Anwendungsprogramms zu verwenden.

Als größte Herausforderung stellt sich die Festlegung des zu verwendenden Referenzmodells für die Schnittstellen heraus. Die volle Bandbreite der heute eingesetzen Schnittstellen-Techniken und Semantiken ist zu heterogen, um sie in einer DSL zusammenzuführen. Wir entscheiden uns für eine Command-Query-Schnittstelle auf Basis asynchroner Nachrichten. Mit der SStDSL werden so die in Projekten häufig neu zu erfindenden Techniken und Semantiken als Best Practices in einer Sprache zusammengeführt.

Im Ergebnis ist die Schnittstellentechnik vom Anwendungsprogramm vollständig entkoppelt. Sie ist ohne Rückwirkung auf das Anwendungsprogramm nicht nur austauschbar, sondern auch in wesentlichen semantischen Aspekten erweiterbar.

Wir implementieren die SStDSL mit dem Parserkombinator PetitParser und demonstrieren an einem Anwendungsbeispiel, dem AuMicrovaweOven, den praktischen Einsatz. Dessen Statechart binden wir mit Hilfe der SStDSL an den zustandslosen Teil der Anwendung an.

1 *Einleitung*

DSLs (Domänenspezifische Sprachen - domain specific languages) sind ein vielversprechender Weg, um in der Programmierung die Abstraktionslücke zu schließen und dem Programmierer problemnahe Sprachen an die Hand zu geben[1,2].

Dies gilt insbesondere, wenn die DSLs eingebettet sind, d.h. ihr Parser und ihr *semantisches Modell*[3] in der Wirtssprache leben[4]. Eine dankbare Problemklasse für eingebettete DSLs sind Konfigurationssprachen, deren Code vor der eigentlichen Ausführungszeit der Anwendung bestimmte Teile des Systems wie beschrieben zusammenbaut.

In einer ersten Erfahrung[5] haben die Autoren gelernt, wie eine DSL für Harel Statecharts (die *AuDSL*) mit Hilfe eines Parserkombinators gebaut werden kann und wie vorteilhaft sich diese AuDSL von der gewöhnlichen Framework-Programmierung unterscheidet. Diese Erfahrungen sollten sich auf andere eingebettete Konfigurations-DSLs übertragen lassen.

Allerdings ist das Referenzmodell der AuDSL, die Harel-Statecharts, gut verstanden und abgegrenzt. Eine DSL für Schnittstellen stellt demgegenüber die zusätzliche Herausforderung, erst einmal ein Referenzmodell für Schnittstellen festzulegen, i.e. deren Semantik zu definieren. Herausforderungen dieser Art dürften für die Programmierung mit DSLs typisch sein, eine fertig definierte

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

[2] Fowler, M.: *Domain-Spezific Languages*. Addison-Wesley, 2010

[3] aus den Klassen des semantischen Modells erzeugt der Parser den Code für die Laufzeitmaschine der DSL

[4] Im Gegensatz zu Fowler schließen wir Grammatik-basierte DSLs von dieser Definition nicht aus.
Fowler, M.: *Domain-Spezific Languages*. Addison-Wesley, 2010

[5] Krasemann, H., J. Brauer und C. Crasemann: *DSL MIT PARSER-KOMBINATOREN: Mit wenig Code zu einer Harel-Statechart-DSL*. OBJEKTspektrum, (04), Juli/August 2011

Semantik wie für die Harel-Statecharts ist eher die Ausnahme[6].

Das Referenzmodell für Schnittstellen schließt die Lücke zwischen einem geeignet definierten API für die Anwendung und den sehr verschiedenen fundamentalen Techniken der Schnittstellenprogrammierung wie entfernte Prozeduraufrufe (RPC), FTP, WebServices oder Nachrichten-Systemen wie JMS.

Es bestimmt einerseits die DSL selbst (Nachrichtentypen und Nachrichtenfolgen, aber auch spezifizierbare betriebliche Eigenschaften) und andererseits einen Teil[7] des semantischen Modells der SStDSL.

Dadurch, dass alle spezifizierbaren Eigenschaften mit der DSL angegeben werden und die Anwendung lediglich auf ein API zugreift, entfällt jeder technische Code im Anwendungsprogramm. Änderungen oder Erweiterungen der DSL-Semantik schlagen sich ausschließlich im semantischen Modell nieder, ggf. auch im DSL-Text, nie aber im Anwendungsprogramm selbst. Damit ist die Anwendung von der Schnittstellentechnik vollständig entkoppelt.

Die Implementierung der SStDSL erfolgte mit dem Parserkombinator PetitParser[8] von Helvetia[9], der in vielen Smalltalk-Dialekten verfügbar ist. Parserkombinatoren stehen in einer zunehmenden Zahl von Programmiersprachen integriert zur Verfügung. In der Java-Welt ragt Scala[10] heraus, das einen integrierten Parserkombinator enthält. Deshalb lassen sich die Ergebnisse problemlos auf industrielle JVM-Umgebungen übertragen[11].

Zur Illustration ziehen wir dasselbe Beispiel heran wie für die AuDSL, nämlich den MicrovaweOven. Alle Zustände des MicrovaweOven sind mit der AuDSL in einem Harel Statechart spezifiziert. Die eigentliche Anwendung, in der dieser Statechart eingebettet ist, beherbergt neben der Benutzungs-Schnittstelle nur noch zustandslose Prädikate für die Guards des Harel Statecharts und Aktionsmethoden für die auszuführenden Aktionen.

Für die Demonstration der SStDSL wird die Beispielanwendung des MicrovaweOvens aufgeteilt, vgl. Abbildung 1. Der Harel-Statechart des MicrowaveOven, definiert mit der AuDSL, lebt in einem Image bzw. Rechnerknoten. In einem zweiten Image bzw. Rechnerknoten lebt die eigentliche, zustandslose Anwendung des MicrowaveOven. Mit der SStDSL werden nun beide Seiten der Schnittstelle spezifiziert, sowohl die Anwendungs-Seite als auch die Statechart-Seite. Diese beiden Spezifikationen sind voneinander unabhängig[12].

Mit dem Ansprechen der Schnittstelle sowohl auf der Anwendungs- als auch auf der Statechart-Seite funktioniert die Beispielanwendung MicrovaweOven jetzt verteilt auf zwei Smalltalk-Images, die auf beliebigen Rechnern laufen können.

2 *Das SStDSL-Referenzmodell für Schnittstellen*

Im Bereich der Schnittstellenprogrammierung gibt es eine große Bandbreite von Semantiken und Techniken. Während z. B. COR-

[6] Die Autoren haben hier sowohl die Rolle des Domänenexperten, der i) die Sprache liefert und ii) weiss, worauf es bei der Definition von Schnittstellen ankommt, als auch die des Softwaretechnikers, der die DSL formalisiert.

[7] Das Referenzmodell lässt noch Spielraum für verschiedene Detail-Semantiken.

[8] 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

[9] Renggli, L.: *Dynamic Language Embedding*. Doktorarbeit, Philosophisch-Naturwissenschaftliche Fakultät der Universität Bern, 2010

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

[11] Gosch, D.: *DSLs in Action*. Manning, 2011

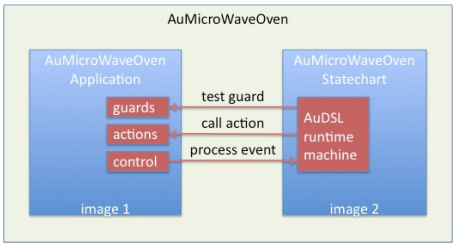

Abbildung 1: Die Verteilung des AuDSL MicrowaveOven auf zwei Knoten

[12] Eine gemeinsame Spezifikation für beide Seiten spart bei einer 2-seitigen Schnittstelle zwar Spezifikationstext, versagt aber bei 3 oder mehr Kommunikationspartnern.

BA[13] und SOAP[14] lediglich darauf abzielen, den objektorientierten Aufruf von Methoden auf das Netzwerk zu verallgemeinern, verknüpft die WSDL-Spezifikation[15] bereits ein Service-Modell[16] mit der Spezifikation von Nachrichten und ihren Signaturen (Inhaltstypen). In allen Fällen werden die Nachrichten-Abfolgen mit den Nachrichtensignaturen, im Fall WSDL auch dem Service-Modell, gemeinsam spezifiziert. Wir erkennen allerdings drei separate Aspekte einer Schnittstelle:

- die Spezifikation der Nachrichten selbst,
- darauf aufbauend die Spezifikation der Nachrichten-Abfolgen (Service bei WSDL),
- und für jede Nachricht die Spezifikation ihres Inhalts (der Typbeschreibung).

In unserem Referenzmodell trennen wir diese drei Aspekte strikt. Die Spezifikation der Nachrichteninhalte erfordert immer eine Konvention zwischen Sender und Empfänger. Im Sinne der losen Kopplung ist dies heute meist nur noch ein Name, nämlich der des beschreibenden XML-Schemas. Die Spezifikation der Nachrichten-Abfolgen hingegen abstrahiert vom Nachrichteninhalt. Wir wählen die Nachrichten und ihre möglichen Abfolgen so, dass die Schnittstellen zwischen Anwendungssystemen (Enterprise-Schnittstellen) damit beschrieben werden können[17]. Insbesondere enthält unser Referenzmodell auch die Nachrichtentypen der WSDL.

Der Referenztext von Hohpe und Woolf[18] beschreibt die Semantik des Nachrichtenaustauschs im wesentlichen auf der Ebene der Kanal- und Nachrichten-Eigenschaften, wie z. B.:

- Nachrichtensysteme und Kanäle als asynchrone Alternative zu synchronen Aufrufschnittstellen;
- die verschiedenen Möglichkeiten für Nachrichtenkanäle wie Punkt-Punkt Verbindungen (*P2P*) oder Rundsendungen (*Broadcast* oder *Publish/Subscribe (Pub/Sub)*);
- Nachrichten mit oder ohne Antwort;
- veschiedenen Nachrichtentypen wie Command-, Document- und Event-Nachrichten[19];
- Eigenschaften von Nachrichtenkanälen wie Persistierung oder Zustellgarantie[20].

Auf dieser Basis ist die Zahl der möglichen Semantiken sehr groß.

Unser Referenzmodell für die Nachrichtentypen und -kanäle soll minimalistisch sein. Gleichzeitig soll es die beteiligten Systeme so weit wie möglich entkoppeln. Drittens soll es auch ermöglichen, die erforderlichen betriebsnahen Eigenschaften inkrementell hinzuzufügen[21].

Weil synchrone RPC-Lösungen[22] zu unnötiger Kopplung der Systeme führen, entscheiden wir uns für asynchrone Nachrichten als Basis der Schnittstelle. Auch dürfen die Nachrichteninhalte ausschließlich Strings sein; Objekte führen mit ihren Typdeklarationen zu unerwünschter Compile- und Build-Abhängigkeit. Wir machen die Spezifikation der betriebnahen Eigenschaften[23] zu einem op-

[13] OMG: *Common Object Request Broker Architecture: Core Specification Version 3.0.3*, März 2004

[14] OMG: *SOAP Version 1.2 Part1: Messaging Framework*, April 2007

[15] OMG: *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*, Juni 2007

[16] Ein *service* der WSDL enthält *interfaces*, die ihrerseits aus Nachrichten-Abfolgen (*operations*) bestehen.

[17] Einer der Autoren baut seit mehr als einem Jahrzehnt die Schnittstellen in Steuerungssystemen nach diesem Modell

[18] Hohpe, G. und B. Woolf: *Enterprise Integration Patterns*. A Martin Fowler Signature Book. Pearson Education, 2004

[19] Bei Hohpe und Woolf unterscheiden sich Event- und Document-Nachrichten i.w. nur in ihrer Häufigkeit

Hohpe, G. und B. Woolf: *Enterprise Integration Patterns*. A Martin Fowler Signature Book. Pearson Education, 2004

[20] wenn der Nachrichtenkanal die Nachricht abgenommen (=quittiert) hat, dann garantiert er, dass er sie dem Empfänger zustellt, sobald dieser für ihn erreichbar ist

[21] Den Wert inkrementeller Spezifikation diskutieren ausführlich

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

[22] auch Java-RMI

[23] dies sind z.Ḃ. Logging, eingebaute Zeitmessungen und ein Anschluss für das Monitoring der Anwendung

tionalen Teil der SStDSL. So kann ihre Spezifikation je nach den Erfordernissen im Entwicklungsprozess später eingebracht werden.

Die minimale Menge der Nachrichtentypen legen wir unter Beachtung der *Command-Query-Separation*[24] wie folgt fest:

[24] siehe Seite 133 in MEYER, B.: *Object-oriented Software Construction*. C.A.R. Hoare Series. Prentice Hall, 1988

2.1 *Nachrichten-Typen*

Nachrichten-Typen und -Abfolgen sind anhand der folgenden Aspekte unterscheidbar:

- Die Art der Verteilung vom Sender zum Empfänger ist entweder 1:1 (P2P) oder 1:n (Pub/Sub);
- eine Nachricht wartet auf Antwort (die Antwort ist synchron) oder nicht;
- 1:1-Nachrichten ohne Antwort und 1:n-Nachrichten sind inhärent asynchron;
- eine Nachricht kann eine Zustellgarantie haben oder nicht;
- eine Nachricht kann eine Zustandsänderung beim Empfänger bewirken oder nicht;
- eine Nachricht ist eine *Triggernachricht*[25] oder eine *Datennachricht*[26].

Wir bilden diese Eigenschaften - außer der Zustellgarantie - auf die folgenden Nachrichtentypen der SStDSL ab:

- Eine *Notification* ist eine asynchrone 1:n-Datennachricht[27]. Der Empfänger aktualisiert seine Datenhaltung mit dem Inhalt der Notification und ändert damit seinen Zustand.
- Ein *Command* ist eine 1:1-asynchrone Triggernachricht, die eine Aktion beim Empfänger auslöst. Damit ändert sie den Zustand des Empfängers.
- Ein *Request* ist eine synchrone 1:1-Triggernachricht ohne Zustellgarantie, die Informationen erfragt und auf die Antwort wartet.[28] Sie ändert beim Empfänger keinen Zustand und kann deshalb beliebig oft wiederholt werden.
- Ein *Reply* ist eine synchrone 1:1-Datennachricht ohne Zustellgarantie als Antwort auf einen Request. Solange keine Zustandsänderung aus anderen Gründen eingetreten ist, ist der Reply auf einen wiederholten Request immer identisch.

Die Zustellgarantie ist als betriebliche Eigenschaft ansonsten unabhängig von den Nachrichtentypen und wird weiter unten getrennt betrachtet.

[25] Eine Triggernachricht enthält ein Semantik tragendes Schlüsselwort (den Trigger) und optional beliebig strukturierte Daten. Bei Hohpe und Woolf ist z.B. *Command* eine Triggernachricht.

[26] Eine Datennachricht enthält beliebig strukturierte Daten, die zugehörige Semantik ist eine Aktualisierung der Daten. Hohpe und Woolf erwähnen die Datennachrichten *Document* und *Event*. Man kann die Datennachricht also auch als Spezialfall der Triggernachricht sehen mit dem weggelassenen Trigger 'Update'.

[27] dies enthält als Spezialfall auch 1:1

[28] Für den Fall, dass die Antwort ausbleibt, hat der Sender eine Fehlerbehandlung. Deshalb ist eine Zustellgarantie überflüssig.

2.2 *Nachrichten-Router*

Im Nachrichten-Austausch heutiger Unternehmens-Software werden in der Regel keine Punkt-zu-Punkt-Verbindungen genutzt, sondern ein Router. Dies hat hauptsächlich zwei Gründe:

- die Zahl der Verbindungen skaliert besser und
- im zentralen Router kann die Dienst-Qualität leicht hergestellt werden.

Das Skalenargument verliert seine Gültigkeit, wenn alle Teilnehmer a priori verbunden sind - wie es im Internet der Fall ist. Dann

ist für Punkt-zu-Punkt-Verbindungen der zentrale Router überflüssig[29]. Die Administration der Kommunikation wird dadurch vereinfacht.

Da es eine Reihe solcher Router fertig verfügbar gibt, kann auch für die Zwecke der SStDSL ein Router genutzt werden. Dieser Router hält für jeden Schnittstellenpartner eine Empfangs-Queue bereit. Die SStDSL kann einen Teil ihrer Funktionalität an den Router auslagern.[30]

Die SStDSL kann verschiedene Router-Technologien benutzen, wie z. B.:

- NONE, oder
- FTP[31], das File Transfer Protocol,
- SQS, die Amazon Simple Queue Services[32],
- AMQP[33], das Advanced Message Queue Protocol, z. B. in Form von RabbitMQ[34] oder
- JMS[35], der Java Message Service.

In der Beispielimplementierung wird ein FTP-Server benutzt.

[29] Jede Schnittstelle enthält dann ihre eigene Empfangsqueue. Für 1:n-Verbindungen reicht dies allerdings nicht aus; diese erfordern mindestens eine zentrale Verwaltung der Abonnements. Auch das Monitoring funktioniert nur über eine zentrale Instanz.

[30] Der Spezifikationstext einer Schnittstelle ist idealerweise unabhängig von der Existenz des Routers.

[31] Für FTP reicht ein FTP Server mit je einem Verzeichnis pro Queue aus.

[32] Amazon: *Amazon Simple Queue Service (Amazon SQS) aws.amazon.com/de/sqs*, 2011

[33] *Advanced Message Queuing Protocol - www.amqp.org*

[34] *RabbitMQ - AMQP Messenger www.rabbitmq.com*

[35] Sun Microsystems: *Java Message Service 1.1*, April 2002

2.3 *Synchronizität*

Um zu verstehen, wie synchrone und asynchrone Kommunikation im Sinne der SStDSL entstehen, betrachten wir die drei obersten Schichten der Kommunikation, vgl. Abbildung 2.

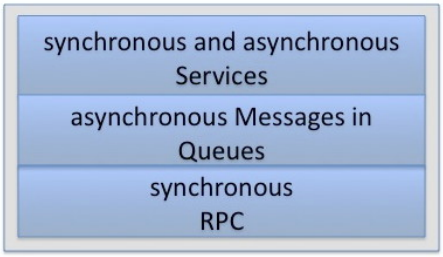

Abbildung 2: Die obersten drei Kommunikationsschichten der SStDSL

Die unterste Schicht arbeitet synchron. So wird z. B. eine Nachricht gesendet, indem sie in ein FTP-Verzeichnis geschrieben oder ein RPC aufgerufen wird. Der Sender wird nach dem FTP- oder RPC-Aufruf fortgesetzt.

Die mittlere Schicht macht durch Einführung der Warteschlangen (Queues) vor den Empfängern aus den synchronen Aufrufen asynchrone Nachrichten. Die Nachricht wird irgendwann (asynchron) der Warteschlange entnommen und startet eine eigene Verarbeitung beim Empfänger.

Auf der obersten Schicht wird die Nachrichten-Abfolge (Service) Request/Reply synchron. Sobald ein Reply eintrifft, wird damit der wartende Request fortgesetzt. Alle anderen Nachrichten bleiben asynchron.[36]

[36] Der Vorteil dieser - auf den ersten Blick umständlich anmutenden - Konstruktion liegt in der Auswechselbarkeit der Implementierungen der mittleren und unteren Schicht.

Die SStDSL implementiert dieses Verhalten durch zwei verschiedene Aufrufe für

- synchrone Queries (Request/Reply) und
- asynchrone Commands und Notifications.

Beim synchronen Senden eines Request antwortet die Laufzeitmaschine mit dem Reply oder einem Timeout-Fehler. Das asynchrone Senden eines Commands kehrt ohne Antwort zurück.

Damit sind die kommunizierenden Anwendungen so weit wie möglich entkoppelt. Die unmittelbare Verfügbarkeit des Kommunikationspartners ist nur noch dort relevant, wo die Anwendung es erfordert. Der Grad der Kopplung ist über die Timeouts spezifiziert.

2.4 *Publish/Subscribe*

Nur Notifications werden an unbekannte Empfänger gesendet (1:n)[37]. Im diesem Fall sendet die Anwendung an ein *Topic*[38], eine spezielle Pub/Sub-Queue. Das Topic wird durch einen zentralen Dienst komplementiert, mit dem jeder beliebige Empfänger bestimmte Notifications abonnieren oder das Abonnement wieder löschen kann. Das Abonnement eines Empfängers lautet auf seine Queue, den Namen einer Notification, und dem Namen ihres Datums als Aspekt[39]. Das Topic stellt abonnierte Nachrichten in der Empfängerqueue des Abonnenten zu und verwirft alle nicht abonnierten Nachrichten.

Für Pub/Sub stellt die SStDSL das API für das Abonnieren und für das Senden von Notifications zur Verfügung. Mit der SStDSL wird die Router-URL spezifiziert, wo die Laufzeitmaschine den zentralen Abonnementsdienst findet und wo ggf. auch das Topic liegt.

[37] Da der Absender die Empfänger nicht kennt, gibt es keine Vereinbarung mit dem Empfänger über die Semantik eines Triggerwortes. Im Zweifel würde der Empfänger weder ein Command noch eine Request verstehen. Notifications hingegen haben die feste Semantik der Daten-Aktualisierung.

[38] hier folgen wir dem Begriff aus der JMS-Spezifikation

[39] Auch der Absender könnte noch Aspekt des Abonnements sein. Jedenfalls stellt dies eine Untermenge der für JMS spezifizierten Funktionalität der *message selectors* dar.

2.5 *Minimalistik: nur eine Empfangsqueue*

In realen Systemen sieht man oft mehrere Empfangsqueues für einen Empfänger. Gründe dafür sind:

- Trennung der synchronen Antworten von asynchronen Nachrichten;
- Trennung von Nachrichtentypen oder
- parallele Verarbeitung auf der Empfängerseite.

Die parallele Verarbeitung auf der Empfängerseite erfordert Reentrancy der Verarbeitung, deshalb beschränken wir uns auf single threaded Empfänger. Die Trennung der synchronen Antworten von asynchronen Nachrichten soll für das Anwendungsprogramm transparent sein. Eine Trennung der Nachrichtentypen ist überflüssig, weil die SStDSL für jede Nachricht eine eigene Verarbeitungsmethode erzeugt. Damit entfallen alle semantischen Gründe für verschiedene Empfangsqueues. Der Router stellt neben der Empfangsqueue der Anwendung ein Topic zur Verfügung, in das Notifications gesendet werden können. Wenn eine Anwendung eine Notification abonniert hat, wird der Router diese in der Empfangsqueue ausliefern.

2.6 *Versionierung*

Wir versionieren sowohl die Schnittstelle selbst aus auch jede einzelne Nachricht.

Die Version der Schnittstelle dient mindestens der Dokumentation, darüberhinaus aber auch der Identifikation einer bestimmten Semantik, nämlich der Bedeutung einzelner Nachrichten oder Nachrichten-Reihenfolgen.

Die Version der einzelnen Nachricht wird verwendet, um die Verarbeitung bestimmter Nachrichten-Versionen zuzusichern. Damit wird es möglich, Schnittstellen gezielt zu migrieren, ohne

Deployment-Abhängigkeiten zu erzeugen. Eine neue Nachrichten-Version (neu bzgl. Signatur oder Semantik) wird zuerst bei den möglichen Nachrichten-Empfängern eingeführt, danach bei den Sendern. Das bedeutet auch, dass jede Schnittstelle i. A. mehr als eine Version einer Nachricht verarbeiten können muss.[40]

Zu einem gegebenen Zeitpunkt wird jede Nachricht einer Schnittstelle in genau einer Version gesendet, aber in möglicherweise mehreren Versionen empfangen.

[40] Die Versionierung schützt nicht vor semantischer Kopplung bei gleichzeitiger Änderung eines Request/Reply-Paares. Es ist Sache der Anwendungsentwicklung, die Semantik schrittweise so zu ändern, dass die beteiligten Systeme nicht gleichzeitig verändert werden müssen.

2.7 *Spezifikation der Schnittstelle (Interface)*

Zu den verschiedenen Aspekten der Spezifikation einer Schnittstelle gehören neben dem Namen und der Version:[41]

- die Schnittstellen-Technik (FTP, SQS, AMQP, JMS oder NONE),
- die Router-Adresse (eine URL),[42]
- die Empfängerqueue (ein Name),
- die Menge der Nachrichten in ihren zeitlichen Abfolgen,
- betriebsnahe Eigenschaften,
- und jede einzelne Nachricht.

[41] Auch wenn kein Router benutzt wird, müssen in der SStDSL die URL der zentralen Instanz und der Name der eigenen Empfangs-Queue spezifiziert werden.

[42] in jedem Fall liegen hier die Abonnements-Dienste.

Die betriebsnahen Eigenschaften sind optional, um sie inkrementell hinzufügen zu können. Zu den betriebsnahen Eigenschaften zählen vor allem das Logging und das Monitoring, aber auch die Zeitmessungen. Für das Logging werden verschiedene Ebenen (Loglevel) im semantischen Modell spezifiziert, die hier angezogen werden, und zwar gleichförmig für alle Nachrichten. Um das Zeitverhalten zu überwachen, bietet es sich an, die Reaktionszeit einer synchronen Antwort zu messen. Solche Zeitmessungen werden im semantischen Modell implementiert und können hier durch Angabe der Nachrichten, deren Antwortszeit gemessen werden soll, vorgegeben werden. Ein optionales zentrales Monitoring sammelt Fehlernachrichten über die Kommunikation auf. Nur wenn es spezifiziert ist, werden die in Abschnitt 4.3 beschriebenen Fehlernachrichten gesendet.

Der Loglevel ist für alle Nachrichten gleich.

Damit haben wir als Spezifikation der Schnittstelle:

Schlüsselwörter der DSL sind in kursiven *GROSSBUCHSTABEN* geschrieben, die bereitzustellende Information ist in < ... > geklammert.

```
INTERFACE  <Schnittstellenname>
   TECHNOLOGY  <FTP, SQS, AMQP, JMS oder NONE>
   URL  <Router-URL>
   VERSION  <Schnittstellenversion>
   SEND  <Liste der Nachrichten, die gesendet werden können - mit Versionen und Sequenzen>
   RECEIVE <Empfangsqueue und Liste/Sequenzen  der Nachrichten, die empfangen werden können>
   TIMING  <ausgewählte Nachrichten>
   {LOGLEVEL  <vordefinierter Loglevel>}
   {MONITORING <Monitorqueue>}
```

und als Spezifikation der Nachrichten:

```
MESSAGES
   (<NOTIFICATION|COMMAND|REQUEST|REPLY> <Nachrichtenname>
      VERSION <Nachrichtenversion>
      DATA <Datentyp>)*
```

INTERFACE und MESSAGES zusammen stellen die SStDSL-Spezifikation dar. Schnittstellenname und -version, die Technologie, Router-URL und Empfangsqueue müssen bereits zu Beginn spezifiziert werden. Alle Nachrichten und die betriebsnahen Eigenschaften können im Laufe der Entwicklung inkrementell hinzugefügt werden.

2.8 *Querschnittliche Information*

Ein zentraler Dienst verwaltet die Abonnements von Notifications. Zu diesem Dienst gehören die Entgegennahme oder Veränderung eines Abonnements, das Speichern des Abonnements, und das Ermitteln aller Empfänger für eine gegebene Notification.

An gleicher Stelle liegt die Monitorqueue.

Wenn ein Router verwendet wird, liegt der zentrale Dienst dort. In dem Fall sind einige der spezifierten Informationen gleichermaßen dort, also doppelt vorhanden. So muss der Router die spezifizierten Empfangsqueues mit der spezifizierten Technik implementieren. Genauso implementiert der Router die Empfangsqueues anderer Kommunikationsteilnehmer. Deren Namen müssen im Anwendungsprogramm bekannt sein.[43] Für die zu sendenden Nachrichten muss die Anwendung also den Empfänger selbst bestimmen.

[43] Die Spezifikation könnte hier allenfalls eine Übersetzung leisten, darauf verzichten wir.

Wenn ein Router verwendet wird, implementiert er außerdem den Topic für Notifications. Der SStDSL ist der Topic-Name bekannt, für die Anwendung ist er transparent.

2.9 *Zustellgarantie und Dauerhaftigkeit*

Ein Nachrichtenkanal kann dem Sender einer Nachricht mit der Sendequittung eine *Zustellgarantie* geben, d. h. dass er die Nachricht so lange versucht zuzustellen, bis der Empfänger sie abgenommen (quittiert) hat.

Während Requests und Replies keine Zustellgarantie brauchen, können Commands mit einer Zustellgarantie versehen werden. Diese Zustellgarantie ist nützlich, wenn es in der Anwendung keinen Buchhaltungsmechanismus gibt, mit dem verloren gegangene Nachrichten ggf. nachgefordert werden können. Die SStDSL speichert dafür die Nachrichten so lange zwischen, bis sie zugestellt sind[44].

[44] Bei Verwendung eines Routers übernimmt der die Zwischenspeicherung. Die Zustellgarantie wird deshalb auch *store and forward* genannt.

Für Pub/Sub von Notifications kann ein Abonnent bei der Anmeldung im Abonnements-API die *Dauerhaftigkeit*[45] verlangen oder nicht. Dies ist die Garantie, abonnierte Nachrichten so lange aufzubewahren, bis sie dem Abonnenten zugestellt werden können. Ohne Dauerhaftigkeit wird eine Notification nur den erreichbaren Empfängern zugestellt. Hierfür braucht nichts spezifiziert zu werden.

[45] *durable subscription*

2.10 Transaktionalität

Darüber hinaus kann das Senden einer Nachricht in dem Sinne transaktional sein , dass sie zusammen mit weiterer Funktionalität innerhalb einer Transaktion ausgeführt wird oder nicht. Hiervon sieht die SStDSL ab.

2.11 Über die Rolle der Commands

Commands können von Anwendung zu Anwendung beliebig zu Sequenzen verkettet werden, als Triggernachricht orchestrieren sie damit die verteilte Verarbeitung z. B. eines Geschäftsprozesses. Diese Sequenzen sind Teil der Anwendungssemantik, die nicht mit der SStDSL spezifiziert werden kann.

In der Praxis werden die zwischen den beteiligten Systemen auftretenden Sequenzen von Commands höchstens teilweise spezifiziert.

Jedes einzelne Command[46] hat die folgenden transaktionalen Eigenschaften: Seine Verarbeitung ist

- atomar;
- konsistent;
- isoliert.

Eine Kette von Nachrichten dagegen kann partiell fehlschlagen und verliert damit die Garantie der Konsistenz. Eine Anwendung mit der SStDSL muss deshalb ihre Transaktionen als Reaktion auf jeweils eine Nachricht ausführen oder zusätzliche Konsistenzmechanismen implementieren.

Häufig kommen Anfragen vor, die im antwortenden Teilsystem Zustand ändern wie z. B. eine Reservierungsanfrage. Solch eine Query kann nicht als Request/Reply, sie muss als Folge zweier Commands oder eines Commands und einer Notification spezifiziert werden.

[46] aber auch jede andere Nachricht

3 Beispiel: Die Statechart-Schnittstelle des MicrowaveOven

Wir rekapitulieren den AuMicrowaveOven und seine Zerlegung auf prinzipiell zwei Rechnerknoten. Sodann beschreiben wir die Spezifikation der beiden Schnittstellen-Teile und den zugehörigen konkreten DSL-Text. Abschließend zeigen wir die von der SStDSL generierten Methoden, mit denen der Programmierer den Anschluss an die neue Schnittstelle herstellt.

3.1 Der MicrovaweOven der AuDSL

Der MicrovaweOven simuliert einen Mikrowellenherd. Im Mikrowellenhert treten folgende Ereignisse auf:

- *start*, wenn der Nutzer den Knopf zum Starten des Kochvorgangs drückt;
- *open* und *close*, wenn der Nutzer die Tür öffnet oder schließt; und
- *finished*, wenn die Kochzeit im eingebauten Wecker abgelaufen ist.

Der Zustandsautomat ruft je nach Zustand Aktionsmethoden auf:

- *enableTimeSetting* und *disableTimeSetting*, um das Einstellen des Weckers zu erlauben oder zu verhindern; und
- *startTimer* und *stopTimer*, um den Kochvorgang zu starten, fortzusetzen oder zu unterbrechen.

Beim Ereignis *start* prüft der Zustandsautomat mit dem Guard *doorIsClosed*, ob die Tür geschlossen ist.

Aus diesen Ereignissen, Aktionen und Prüfungen wird eine Schnittstelle, wenn wir den Mikrowellenofen in den Zustandsautomaten (MicrowaveStatechart) selbst und den Rest der Anwendung (MicrowaveApp) zerlegen.

Daraus folgen diese generellen Nachrichten von der MicrowaveApp zum MicrowaveStatechart

- *process: anEvent*, um ein Ereignis auszuführen;

und vom MicrowaveStatechart zur MicrowaveApp:

- *predicate: aGuard*, um einen Guard zu testen; und
- *action: anAction*, um eine Aktion auszuführen.

Für Test- sowie Debugging-Zwecke benötigen wir zwei weitere Nachrichten, nämlich

- *print*, um den Zustand in druckbarer Form zu erfahren; und
- *isEnteredAt*, um einen Zustand zu testen.

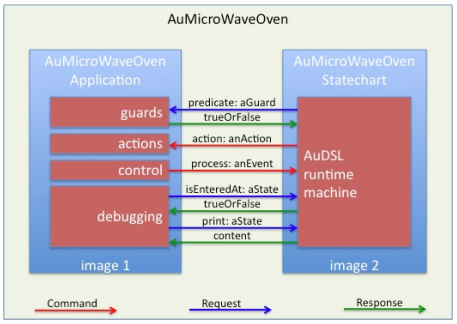

Abbildung 3: Die Kommunikation zwischen AuDSL MicrowaveOvenApp und MicrowaveStatechart

3.2 *Die INTERFACE Spezifikation*

Jetzt können wir den SStDSL-Text für die Schnittstelle zwischen der MicrowaveOven Anwendung und dem zugehörigen Statechart hinschreiben.

Diese Schnittstelle wird für jede Seite[47] einzeln vollständig spezifiziert. Der INTERFACE-Teil der Spezifikation[48] zitiert die Nachrichten nur mit Namen und Version. Die Nachrichten selbst werden im Teil MESSAGES spezifiziert.

[47] Eine gemeinsame Spezifikation für beide Seiten spart bei einer 2-seitigen Schnittstelle zwar Spezifikationstext, versagt aber bei 3 oder mehr Kommunikationspartnern.

[48] Die in kursiven *GROSSBUCHSTABEN* geschriebenen Wörter sind Schlüsselwörter der DSL.

```
INTERFACE MicrowaveStatechart /* bei der MicrowaveApp */
   TECHNOLOGY FTP
   URL www.myhomepage.com
   VERSION 1.0
   SEND (
      COMMAND process VERSION 1.0
      REQUEST print VERSION 1.0
         REPLY content VERSIONS [1.0 0.9] TIMEOUT 500
      REQUEST isEnteredAt VERSION 1.0
         REPLY trueOrFalse VERSIONS [0.9 1.0] TIMEOUT 600 )
   RECEIVE toApp (            /* toApp is the receiver queue */
      COMMAND do VERSIONS [0.9 1.0]
      REQUEST test VERSIONS [0.9 1.0]
         REPLY trueOrFalse VERSION 1.0 )
   TIMING ( )

INTERFACE MicrowaveApp /* beim MicrowaveStatechart */
   TECHNOLOGY FTP
   URL www.myhomepage.com
```

```
   VERSION 1.0
   SEND (
      COMMAND do VERSION 1.0
      REQUEST test VERSION 1.0
         REPLY trueOrFalse VERSION [0.9 1.0] TIMEOUT 400 )
   RECEIVE toSC (            /* toSC is the receiver queue */
     COMMAND process VERSIONS [1.0]
     REQUEST print VERSIONS [0.9 1.0]
         REPLY content VERSION 1.0
     REQUEST isEnteredAt VERSIONS [0.9 1.0]
         REPLY trueOrFalse VERSION 1.0 )
   TIMING ( )
```

Die Nachrichten sind:

```
MESSAGES (
   COMMAND process
      VERSION 1.0
      DATA event
   COMMAND do
      VERSION 1.0
      DATA actionName
   REQUEST test
      VERSION 1.0
      DATA predicateName
   REPLY trueOrFalse
      VERSION 1.0
      DATA trueOrFalse
   REQUEST isEnteredAt
      VERSION 1.0
      DATA stateName
   REQUEST print
      VERSION 1.0
      DATA scName
   REPLY content
      VERSION 1.0
      DATA contentString )
```

Wir sehen, dass beim Sender die verwendete Version jeder Nachricht spezifiziert ist, z. B.

```
COMMAND process VERSION 1.0.
```

und beim Empfänger die zulässigen Versionen, z. B.:

```
REQUEST print VERSION 1.0
   REPLY content VERSIONS [1.0 0.9] TIMEOUT 500
```

Auf die Spezifikation von LOGLEVEL und MONITORING verzichten wir zu diesem Zeitpunkt noch.

3.3 Aufrufschnittstelle

Aus je einer Schnittstellen- und Nachrichten-Spezifikation werden beim Parsen[49] die Laufzeitmaschine und ihr API erzeugt. Das API besteht in unserem Fall aus Smalltalk-Methoden.

Der Parser meldet einen Fehler, wenn eine Nachricht nicht im Abschnitt MESSAGES spezifiziert wurde aber im Abschnitt INTERFACE vorkommt.

Die mit den obenstehenden MESSAGES aus dem INTERFACE MicrovaweApp erzeugten API-Methoden sind:

[49] Mit Parserkombinatoren wie dem PetitParser werden beim Parsen bereits alle Produktionen des Compilers ausgeführt.

```
MicrowaveApp methodsFor: 'SStDSL sends'
process: anEvent at: aReceiver
   "(command) generated by SStDSL
   call this method from your code - do not change"
   self send: (self newMsg: #process content: anEvent)
        to: aReceiverQueue

MicrowaveApp methodsFor: 'SStDSL requests'
isEnteredAt: aStateName at: aReceiver
   "(request) generated by SStDSL - answers the reply content of
   message #trueOrFalse: aTrueOrFalse
   call this method from your code - do not change"
   ^self request: (self newMsg: #isEnteredAt content: aStateName)
         to: aReceiverQueue
print: aScName at: aReceiver
   "(request) generated by SStDSL - answers the reply content of
   message #content: aContentString
   call this method from your code - do not change"
   ^self request: (self newMsg: #print content: aScName)
         to: aReceiverQueue

MicrowaveApp methodsFor: 'SStDSL receive hooks'
answerTest: aPredicateName
   "(request) hook method generated by SStDSL"
   "please define your code - answer aTrueOrFalse "
   ^'exampleString'
doDo: anActionName
   "(command) hook method generated by SStDSL"
   "please define your code"
```

Hier haben wir die Smalltalk-Methodenkategorien mit angegeben, weil sich daraus bereits die Verwendung der Methoden erkennen lässt. *Receive hooks* sind alle Methoden, die für den Empfang der Nachrichten ausprogrammiert werden müssen, *sends* und *requests* sind Methoden, die zum Senden aufgerufen werden. Daneben gibt es noch einige den Programmierer nicht interessierende *private* Methoden sowie das – hier nicht benutzte – API zum Abonnieren von Notifications und zum Senden von Fehlermeldungen an das Monitoring.

Die aus den MESSAGES und dem INTERFACE Microvawe-

Statechart erzeugten API-Methoden sind:

```
MicrowaveStatechart methodsFor: 'SStDSL sends'
do: anActionName at: aReceiver
   "(command) generated by SStDSL
   call this method from your code - do not change"
   self send: (self newMsg: #action content: aMethodName)
        to: aReceiverQueue

MicrowaveStatechart methodsFor: 'SStDSL requests'
test: aPredicateName at: aReceiver
   "(request) generated by SStDSL - answers the reply content of
   message #trueOrFalse: aTrueOrFalse
   call this method from your code - do not change"
   ^self request: (self newMsg: #predicate content: aMethodName)
         to: aReceiverQueue

MicrowaveStatechart methodsFor: 'SStDSL receive hooks'
answerIsEnteredAt: aStateName
   "(request) hook method generated by SStDSL"
   "please define your code - answer aTrueOrFalse "
   ^'exampleString'
answerPrint: aScName
   "(request) hook method generated by SStDSL"
   "please define your code - answer aContentString "
   ^'exampleString'
doProcess: anEvent
   "(command) hook method generated by SStDSL"
   "please define your code"
```

Eine bereits vorhandene Methode der *receive hooks* wird von der SStDSL nicht ersetzt. Damit geht der ergänzte Code nicht verloren. Die Ausprogrammierung würde in Smalltalk z. B. so aussehen:

```
MicrowaveApp methodsFor: 'SStDSL receive hooks'
doDo: anActionName
      "(command) hook method generated by SStDSL"
      self perform: anActionName asSymbol
answerTest: aPredicateName
      "(request) hook method generated by SStDSL"
      "please define your code - answer aTrueOrFalse "
      ^self perform: aPredicateName asSymbol
```

4 *Die SStDSL*

Die SStDSL besteht aus dem Parser und dem semantischen Modell einschließlich einer API-Klasse. Der Parser benutzt das semantische Modell für seine Produktionen. In der API-Klasse oder einer Unterklasse davon werden die Methoden zum Senden und Empfangen von Nachrichten generiert.

4.1 Der Parser der SStDSL

Der Parser für die SStDSL wurde mit dem PetitParser[50] implementiert. In Helvetia[51] ist der PetitParser enthalten (er ist dort der Standard-Parser), in Pharo muss er nachinstalliert werden.

Die Grammatiken für die Interface-Spezifikation INTERFACE und für die Nachrichtenspezifikation MESSAGES werden zusammengefügt, weil ein Teil der Generierung, nämlich die Erzeugung der API-Methoden, beide Informationen gleichzeitig benötigt. Damit lautet die Grammatik, unter Verwendung von vielen Schlüsselwörtern etwas langatmig aber lesbar und vollständig hingeschrieben:

[50] 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

[51] Renggli, L.: *Dynamic Language Embedding*. Doktorarbeit, Philosophisch-Naturwissenschaftliche Fakultät der Universität Bern, 2010

```
sstDSL = interface | messages
/* ======================== Parser für das INTERFACE */
interface = 'INTERFACE' identifier
            'TECHNOLOGY' technology
            'URL' url
            'VERSION' version
            'SEND' send*
            'RECEIVE' queue2app receive*
            'TIMING'  timings
            {'LOGLEVEL'  logExceptions | logMessages | logFields}
            {'MONITORING' monitorQueue}
technology = ftp | jms | amqp | sqs
ftp = 'FTP' /* und entsprechend für die anderen Technologien */
url = (letter | '.' | '/')+
send = sendCommand | sendNotification | sendRequestReply
sendCommand = 'COMMAND' identifier 'VERSION' version
sendNotification  = 'NOTIFICATION' identifier 'VERSION' version
sendRequestReply = sendRequest receiveReply timeout
sendRequest = 'REQUEST' identifier 'VERSION' version
receiveReply = 'REPLY' identifier 'VERSIONS' versions
timeout = 'TIMEOUT' integer
queue2app = queue
receive = receiveCommand | receiveNotification | receiveRequestReply
receiveCommand  = 'COMMAND' identifier 'VERSIONS' versions
receiveNotification = 'NOTIFICATION' identifier 'VERSIONS' versions
receiveRequestReply = receiveRequest sendReply
receiveRequest = 'REQUEST' identifier 'VERSIONS' versions
sendReply = 'REPLY' identifier 'VERSION' version
timings = '(' methodName* ')'
methodName = identifier
logExceptions = 'exceptions'
logMessages = 'messages'
logFields = 'fields'
monitorQueue = queue
/* ======================== Parser für die MESSAGES */
messages = 'MESSAGES' message+
message = command | notification | request | reply
```

```
command = 'COMMAND' identifier 'VERSION' version 'DATA' identifier
notification = 'NOTIFICATION' identifier 'VERSION' version 'DATA' identifier
request = 'REQUEST' identifier 'VERSION' version 'DATA' identifier
reply = 'REPLY' identifier 'VERSION' version 'DATA' identifier
/* ======================== gemeinsam genutzte Parser */
queue = identifier
versions = '[' version+ ']'
version = (letter | digit | '.')+
integer = ziffer+
identifier = letter word*
```

Diese Grammatik lässt sich bereits aus dem Beispiel von Abschnitt 3.2 ablesen. Sie wird mit wenigen syntaktischen Modifikationen in den entsprechenden Smalltalk-Code des PetitParser übertragen, so wie es die Autoren anhand des AuDSL-Beispiels[52] gezeigt haben.

[52] Krasemann, H., J. Brauer und C. Crasemann: *DSL MIT PARSER-KOMBINATOREN: Mit wenig Code zu einer Harel-Statechart-DSL.* OBJEKTspektrum, (04), Juli/August 2011

4.2 *Das semantische Modell*

Um aus einem SStDSL-Text eine Laufzeitmaschine mit funktionierendem API erzeugen zu können, braucht der Parser das semantische Modell. Das semantische Modell besteht im Wesentlichen aus den Klassen

- SStDSLInterface,
- FTPSenderReceiver (und weitere für andere Verbindungstechniken),
- SStDSLMsgRecord mit verschiedenen Unterklassen,
- SStDSLMessage mit verschiedenen Unterklassen, und
- SStModel.

Die SStDSLMessage besteht aus dem eigentlichen Nachrichteninhalt, dem *content* und Umschlagsdaten wie dem Nachrichtentyp und -namen, der Version, der MessageID und der CorrelationID. Die *CorrelationID* eines Reply ist die MessageID des auslösenden Requests. Die Laufzeitmaschine kennt diese MessageID während der Verarbeitung des Requests und kann sie deshalb als CorrelationID in den Reply einsetzten.

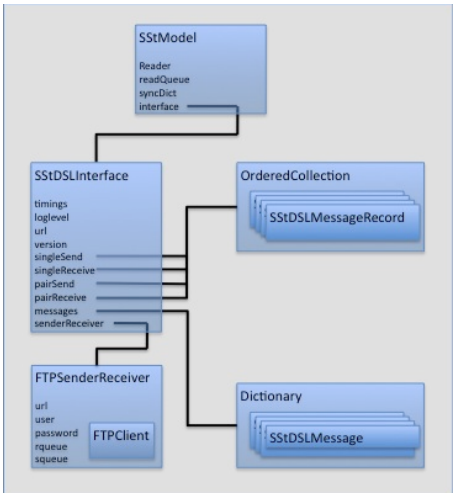

Abbildung 4: Die Klassenstruktur des semantischen Modells

Umschlagsdaten und Nachrichteninhalt werden von der Laufzeitmaschine in eine Draht-Repräsentation übersetzt, das ist in unserem Fall eine Datei mit Namen und mehrzeiligem Inhalt, je eine Zeile pro Umschlagsdatum und eine für den Inhalt. Um nun aus Sicht der Anwendung mehrzeiligen Inhalt zuzulassen, werden die Zeilenwechsel des Nachrichteninhalts für die Dateirepräsentation durch andere Zeichen ersetzt. Die vorliegende Implementierung nutzt für einen Zeilenwechsel die Zeichenkette '«»'. Seitenwechsel sind nicht erlaubt.

Das SStModel verbindet die Anwendung mit der Laufzeitmaschine. Es wird vom eigentlichen Anwendungsprogramm benutzt oder die Anwendung leitet gar davon ab.

Das SStModel hält ein SStDSLInterface. Dort ist die geparste Struktur der Schnittstelle abgelegt. Das SStDSLInterface merkt sich in einem Dictionary alle spezifizierten Nachrichten als Ko-

piervorlage (SStDSLMessages) und mit Hilfe der verschiedenen SStDSLMsgRecords die zu empfangenden und zu sendenden Nachrichten. Es hält außerdem einen SenderReceiver, in unserem Fall einen FTPSenderReceiver, der seinerseits einen FTPClient nutzt.

Der SenderReceiver initiiert beim Programmstart einen Thread, der die Empfangsqueue regelmäßig abfragt und die Nachrichten dort einzeln ausliest. Dieser Thread

- verarbeitet einen Reply sofort mit Hilfe des Synchronisierungs-Dictionarys oder syncDict des SStModel[53], in dem alle offenen Requests gespeichert sind[54];
- stellt alle anderen Nachrichten in die readQueue des SStModel (in der zeitlichen Reihenfolge des Sendestempels);
- und löscht die gelesene Nachricht im FTP-Verzeichnis, unabhängig davon, ob sie verarbeitet wurde oder zu einem Fehler führte.

Der Reader des SStModel arbeitet die ReadQueue ab und führt jeweils die der Nachricht zugehörige Empfangsmethode aus. Es gibt genau einen Reader-Thread, damit die Empfangsmethoden nicht parallel ausgeführt werden[55].

Im SStModel findet man auch die generierten API-Methoden, mit denen die Anwendung senden kann und empfängt. Diese Methoden sind in vier Kategorien organisiert, vgl. Abschnitt 3.3:

- SStDSL receive hooks,
- SStDSL sends,
- SStDSL requests, und
- SStDSL private.

Die eigentlich empfangenden Methoden sind *SStDSL private*, sie werden von der Laufzeitmaschine aufgerufen[56]. Sie rufen ihrerseits generierte *SStDSL receive hooks* Methoden auf, die vom Programmierer ergänzt werden müssen.

Die Methoden in *SStDSL sends* und *SStDSL requests* werden vom Anwendungsprogramm aufgerufen. An ihnen sind Änderungen tabu. Für die *SStDSL requests* erledigt die Laufzeitmaschine wie beschrieben die Synchronisation mit dem Reply, die Methoden antworten mit dem jeweiligen Inhalt des Replies.

Es kann vorkommen, dass eine Nachricht empfangen wird, die nicht spezifiziert ist. Nicht spezifizierte Nachrichten werden grundsätzlich konsumiert[57] und ignoriert. Gleiches gilt für einen Reply, der zu spät kommt, weil der Timeout abgelaufen ist. Die Laufzeitmaschine meldet in solchen Fällen einen Fehler an den unter *MONITORING* genannten Empfänger, sofern das MONITORING spezifiziert ist.

Jede empfangene Nachricht wird auf Versionskompatibilität überprüft. Sie wird nur bei Kompatibilität verarbeitet. Fehler werden ebenfalls dem Monitoring gemeldet.

Der FTPSenderReceiver sendet Nachrichten, in dem er sie als Textdateien in ein FTP-Verzeichnis, die Queue des fremden Empfängers, legt[58]. Der Name einer solchen Datei muss den Konventionen der Hilfsklasse FTPFilename gehorchen. Insbesondere besteht

[53] Diesem Lese-Thread wird eine Lesefunktion des SStModel als Closure übergeben. die in ihrem Kontext das SyncDict kennt.

[54] Im SyncDict wurde beim Senden die Message-ID der ausgehenden Nachricht zusammen mit einer Semaphore gespeichert, an der der Sender wartet. Der lesende Thread ermittelt beim Empfang eines Reply aus dessen Correlation-ID den ursprünglichen Request und speichert die Antwort wiederum im SyncDict ab. Dann wird der ursprüngliche Sender fortgesetzt und kann die Antwort auslesen

[55] sonst müsste die Anwendung eine Garantie geben, dass alle Empfangsmethoden reentrant sind

[56] Den Programmierer interessieren sie nicht, aber leider kann man sie in Smalltalk nicht ganz verstecken

[57] um keinen Speicher zu „fressen“

[58] Je Queue stellt der FTP-Router genau ein Verzeichnis zur Verfügung

er aus der MessageID mit dem Zeitstempel und dem Nachrichten-Namen.

Zum Empfangen pollt der FTPSenderReceiver das FTP-Verzeichnis der eigenen Empfangs-Queue und liest die Nachrichten in Form eben solcher Textdateien.

Der FTPSenderReceiver loggt jedes Senden und Empfangen einer Nachricht.

4.3 Fehlerbehandlung

Wir unterscheiden nach Fehlern beim Parsen, nach syntaktischen und semantischen Laufzeitfehlern. Sodann diskutieren wir den Meldeweg für die Fehler.

Parsefehler

Zur Parsezeit kann der SStDSL-Text fehlerhaft sein oder es tritt in einer Produktion ein Fehler im Parser bzw. in der Laufzeitmaschine auf. In beiden Fällen generiert der Parser Fehlermeldungen. Im ersten Fall muss der SStDSL-Text geändert werden, im zweiten Fall das semantische Modell.

Laufzeitfehler

Beim Empfangen und Senden einer Nachricht können folgende Fehler auftreten, die unabhängig von allfälligen Exceptions von der Laufzeitmaschine an das Monitoring[59] gemeldet werden:

- Das Senden der Nachricht schlägt fehl, z. B. weil der Nachrichten-Server nicht antwortet (bzw. nicht in das FTP-Verzeichnis geschrieben werden kann).
- Es werden Nachrichten empfangen, die nicht verarbeitet werden können. Dafür kann es mehrere Gründe geben:
  - Der Name der zu lesenden Datei ist kein zulässiger FTP-Filename;
  - die Nachricht ist nicht lesbar (die Konvertierung in das interne Format schlägt fehl);
- es gibt keinen offenen Request mehr zur CorrelationID eines Replies[60];
- eine Nachricht kann nicht verarbeitet werden (die Spezifikation kennt keine Nachricht mit der gefundenen Signatur).

Wenn die Verarbeitung einer Nachricht im Anwendungsprogramm zu einem Fehler führt, dann informiert die Anwendung das Monitoring durch eine geeignete Fehlernachricht. Hierbei ist folgendes anzumerken:

- Von der Empfänger-Anwendung werden Requests korrekt beantwortet oder gar nicht.
- Die Empfänger-Anwendung darf eine Notification oder ein Command nur korrekt und ganz oder gar nicht verarbeiten (transaktionales Verhalten).

Für die beiden Fehlerquellen gibt es zwei verschiedene Fehlernachrichten:

[59] wenn es denn spezifiziert ist

Auftretende Fehler müssen protokolliert werden, um Problemen im Code der Anwendung auf die Spur zu kommen. Dazu dient das Monitoring.

[60] Der Sender hatte zuvor einen Timeout. Zu einer sinnvollen Reaktion der Anwendung gehören i) eine angemessene Zahl von Wiederholungen des Sendeversuchs und ii) eine alternative Verarbeitungslogik nach dem endgültigen Fehlschlag (graceful degradation).

- eine Nachricht *invalidMessage* der Laufzeitmaschine (Timeout, Versionsfehler, etc.) mit der Fehlermeldung aus der SStDSL, die eine Zusammenfassung der auslösenden Nachricht enthält, und
- eine Nachricht *errorMessage* der Anwendung, die eine von der Anwendung erzeugte Fehlermeldung enthält.

Für das Senden der errorMessage stellt die SStDSL ein API zur Verfügung.

4.4 Logging und Zeitmessung

Die Loglevels und Zeitmessungen werden optional mit der SStDSL spezifiziert. Die Loglevel sind *exceptions*, *messages*, *fields*. Zur Laufzeit wird der spezifizierte Loglevel angewendet:

- *exceptions*: nur die Fehler (Exceptions und Fehlernachrichten an das Monitoring);
- *messages*: Fehler und alle Nachrichten mit ihren envelope-Daten, ohne den Content;
- *fields*: Fehler und alle Nachrichten mit allen Feldern.

Alle Fehlerlogs haben eine endliche Größe und werden round robin geschrieben. So sind Aufräumarbeiten überflüssig.

Zeitmessungen werden vom Server für die spezifizierten Nachrichten durchgeführt[61]. Die Auswertung ist fest vorgegeben mit Minimum, Maximum, Mittelwert und Medianwert.

[61] Die Liste der spezifizierten Nachrichten kann leer sein

5 Erweiterungen der SStDSL

Manche Erweiterungen erfordern eine Änderung des API. So wäre es z. B. möglich, für Notifications im Publish/Subscribe-Verfahren zusätzliche Attribute zu definieren, nach denen der Abonnent bei der Anmeldung filtern kann. Dies würde eine Änderung/Erweiterung des Sende-API für Pub/Sub erfordern.

Semantik-Erweiterungen erfordern oft eine Erweiterung der SStDSL Syntax und damit des Parsers. Z. B. gilt dies für Erweiterungen der Spezifikationen des Logging oder der Zeitmessungen. Oder in die Laufzeitmaschine werden zusätzliche Messungen eingebaut, die eine Spezifikation erfordern.

Aber viele Semantik-Änderungen der SStDSL sind auch ohne Änderung des Parsers möglich. Ein Beispiel: Die Laufzeitmaschine könnte z. B. für die Replies jeweils eine eigene Queue verwenden. Im Request wird diese Antwort-Queue vermerkt. Die antwortende Anwendung sendet den Reply an die im Request angegebene Queue.

Eine andere transparente Semantikänderung ist z. B. das Nachvollziehen der Nachrichten-Diskurse oder Sequenzen mit Hilfe der CorrelationID. Die Verarbeitung einer Nachricht im Anwendungsprogramm kann in einem Kontext erfolgen, den die SStDSL beim Empfang erzeugt. Dieser Kontext merkt sich die MessageID der empfangenden Nachricht. Beim Senden einer Nachricht setzt die Laufzeitmaschine die MessageID des Kontextes für jede gesendete

Nachricht als CorrelationID ein. Damit lässt sich aus einem Log des Routers durch Korrelation der MessageIDs und CorrelationIDs zu jeder Nachricht ein Sequenzdiagramm des stattgefundenen Diskurses extrahieren. Solch ein Sequenzdiagramm ist ein wichtiges Element für das Verständnis eines Programm und damit für seine Dokumentation.

6 Erfahrungen

Die SStDSL zwingt zur Festlegung einer Standard-Semantik für Schnittstellen. Ohne diese – nicht unwesentliche – Arbeit lässt sich die SStDSL nicht bauen. Dies dürfte für die meisten anderen Einsatzbereiche von DSLs ähnlich sein.

Die SStDSL zwingt – wie zuvor bereits die AuDSL – zur strikten Trennung der Anwendungsaspekte von den Schnittstellenaspekten. Das äußert sich besonders in folgendem:

- Das API der Anwendung für die Schnittstelle ist minimal in dem Sinne, dass es nur semantisch erforderliche Methoden und semantisch erforderliche Namen in den Methodensignaturen enthält.
- Die Schnittstellenspezifikation ist rein deklarativ.
- Die Lesbarkeit der Schnittstellenspezifikation ist deutlich besser als sie es in der Wirtssprache wäre.
- Die Anwendung enthält keinen technischen Schnittstellen-Code.
- Technische Erweiterungen der Schnittstelle wie Erweiterungen des Monitoring oder die Extraktion von Sequendiagrammen geschehen für die Anwendung vollkommen transparent.
- Für Erweiterungen der Schnittstellensemantik wie z. B. die Erweiterung der filterbaren Attribute von Notifications reicht in vielen Fällen eine Erweiterung des API aus, so dass existierende Anwendungen keinen Änderungsbedarf haben.

Dabei ist der Konzeptions- und Implementierungs-Aufwand der SStDSL nicht viel größer als eine direkte Implementierung im Zuge der Anwendungsprogrammierung. Im Gegenteil, bei großen Projekten mit vielen Schnittstellen dürfte sich die Einfachheit des API schnell auszahlen und die SStDSL sogar die Produktivität des gesamten Teams erhöhen.

Anders als bei der AuDSL gibt es für die SStDSL kein allgemeingültiges Referenzmodell. Der überwiegende Teil der Arbeit steckt dementsprechend in der Auswahl und Festlegung der gewünschten Semantik der SStDSL. Dies dürfte in der Praxis anderer DSLs durchaus ähnlich sein: Der größte Teil der Arbeit geht in die Festlegung der Semantik. Mit der SStDSL haben wir uns festgelegt auf:

- eine auswechselbare Transportschicht,
- asynchrone Nachrichten auf der Transportebene, und
- vier Nachrichtentypen für eine typische Command-Query Schnittstelle.

Auf der anderen Seite zeigt sich hierin gerade der Vorteil des Programmierens mit DSLs. Es geht nicht, ohne dass man sich gründli-

che Gedanken über die Semantik macht. Die syntaktische Separation zwischen Anwendung und DSL zwingt dazu. Für Ungenauigkeiten ist wenig Raum.

Dagegen ist die Implementierung der Grammatik sehr einfach. Obwohl die Grammatik recht groß ist, kann sie schon fast vollständig aus dem ersten Beispiel abgeleitet werden. Genauso einfach ist das Übersetzen der Grammatik in PetitParser-Code. Die Produktionen orientieren sich sehr stark an den Strukturen der INTERFACE-Spezifikation.

Das API umfasst für jeden zu sendenden und zu empfangenden Nachrichtentyp etwa eine Methode.

Es zeigte sich, dass der umfangreiche Unit-Test der SStDSL mit beiden Seiten einer Schnittstelle aus einem Image heraus möglich ist. Der Test sendet zu empfangende Nachrichten – mit Hilfe eines Test-API – einfach am Anwendungs-API vorbei in die Empfangsqueue.

Die Codegröße der SStDSL[62] ist:

- Grammatik: 73 LOC
- semantisches Modell und API: 521 LOC
- Unit-Tests: 15 Tests mit 387 LOC
- die *INTERFACE* Spezifikation: 17 bzw. 18 LOC
- die *MESSAGES* Spezifikation: 22 LOC
- AuMicrovaweOven-Beispiel: 80 LOC

[62] in www.squeaksource.org
Krasemann, H.: *SStDSL-hk.44 www.squeaksource.org/SStDSL.html*, 2012

7 *Ausblick*

Die SStDSL ist in einem fast industrietauglichen Zustand. Die wesentlichen Einschränkungen mit dem Stand dieses Papiers sind

- Smalltalk als Implementierungssprache ist zwar mächtig, aber nicht verbreitet;
- zur Zeit ist nur die FTP-Technologie implementiert, die genannten anderen Transporttechniken wie Amazon-SQS, AMPQ oder JMS sind ebenfalls industrierelevant.

Die Autoren erwägen ihren Einsatz in einem industriellen B2B Anwendungssystem mit Produktcharakter. Dieses Produkt ist in Java implementiert. Ein Zukunftsthema ist damit die Portierung auf eine industriell relevante Plattform. Ein anderes Thema ist die Integration mit Legacy-Systemen. Ein drittes Thema ist die Typisierung der Nachrichteninhalte. In dem Beispiel dieses Papiers sind wir mit einfachen Strings ausgekommen. Schließlich kann man die SStDSL natürlich auch ohne zentralen Router bauen.

- Portierung nach Scala: Scala ist eine Sprache, die – als Java-Superset – auf der JVM läuft und dazu ein Parserkombinator-Framework enthält, mit dem die SStDSL problemlos – genauso wie in diesem Papier beschrieben – implementiert werden kann. Selbst wenn Scala sich nicht als Java-Ersatz durchsetzt, macht die Ergänzung der Java-Entwicklungsumgebung um Scala Sinn. Die DSL-Autoren eines Teams können Scala benutzen, alle anderen Team-Mitglieder können in ihrem vertrauten Java-Kontext

bleiben[63].

- Legacy-Integration: Wenn eine Anwendung mit der SStDSL gebaut werden soll, aber die Gegenseite ein Legacy-System ist, dann muss es Unterstützung geben, eine SStDSL-Schnittstelle auch von Hand zu implementieren. Hierfür ist es nützlich die INTERFACE und MESSAGES Spezifikationen als xml-Dokumente exportieren zu können.
- Typisierung der Nachrichten-Inhalte: Jeder Nachrichteninhalt ist ein String und wird mit einen Namen spezifiziert. Dieser Name trägt die Bedeutung und eine Typinterpretation für den Inhalts-String. Für solch eine Typisierung ist z. B. xsd:schema geeignet, dann sind die Nachrichten-Inhalte xml-Dokumente und der Name bezeichnet das Schema.

  Alternativ könnte man eine weitere DSL für die Beschreibung der Typen des Nachrichteninhalts konzipieren. Diese Typ-DSL wäre orthogonal zur SStDSL.
- Wenn die Anwendungen in einem social media Kontext leben, wird ein Router entbehrlich. Die Verwaltung der teilnehmenden Anwendungen kann in eben diesem Kontext vorgenommen werden, vgl. Abschnitt 5. Damit wird die Administration des Nachrichtenverkehrs einfacher.

[63] Gosch, D.: *DSLs in Action*. Manning, 2011

8 Danksagung

Die Autoren danken Alexander Hagemann für seine umfassende Kommentierung einer früheren Version dieses Papiers.

Literatur

[1] *Advanced Message Queuing Protocol - www.amqp.org*.

[2] *RabbitMQ - AMQP Messenger www.rabbitmq.com*.

[3] Amazon: *Amazon Simple Queue Service (Amazon SQS) aws.amazon.com/de/sqs*, 2011.

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

[5] Fowler, M.: *Domain-Spezific Languages*. Addison-Wesley, 2010.

[6] Gosch, D.: *DSLs in Action*. Manning, 2011.

[7] Hohpe, G. und B. Woolf: *Enterprise Integration Patterns*. A Martin Fowler Signature Book. Pearson Education, 2004.

[8] Krasemann, H.: *SStDSL-hk.44 www.squeaksource.org/SStDSL.html*, 2012.

[9] Krasemann, H., J. Brauer und C. Crasemann: *DSL MIT PARSER- KOMBINATOREN: Mit wenig Code zu einer Harel-Statechart-DSL*. OBJEKTspektrum, (04), Juli/August 2011.

[10] Meyer, B.: *Object-oriented Software Construction*. C.A.R. Hoare Series. Prentice Hall, 1988.

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

[12] OMG: *Common Object Request Broker Architecture: Core Specification Version 3.0.3*, März 2004.

[13] OMG: *SOAP Version 1.2 Part1: Messaging Framework*, April 2007.

[14] OMG: *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*, Juni 2007.

[15] Renggli, L.: *Dynamic Language Embedding*. Doktorarbeit, Philosophisch-Naturwissenschaftliche Fakultät der Universität Bern, 2010.

[16] 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.

[17] 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.

[18] Sun Microsystems: *Java Message Service 1.1*, April 2002.