Diplomarbeit

Entwicklung eines jABC Plugin zur grafischen Beschreibung von Transformationen strukturierter Daten

Johannes Neubauer

22. April 2008

Institution

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl 5 für Programmiersysteme

Gutachter

Prof. Dr. Bernhard Steffen
Dipl. Inform. Ralf Nagel

Hiermit erkläre ich, Johannes Neubauer, dass ich die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Zitate habe ich stets kenntlich gemacht.

Dortmund, den 22. April 2008

____

Johannes Neubauer

Inhaltsverzeichnis

1 Motivation **1**

2 Grundlagen **7**

2.1 Konventionen . . . 7
2.2 jABC . . . 7
2.2.1 Lightweight Process Coordination . . . 8
2.2.2 Service-Independent-Building-Blocks . . . 10
2.2.3 Entwurfsmuster für ausführbare SIBs . . . 12
2.2.4 SIB-Graphen . . . 14
2.2.5 Projekte . . . 15
2.2.6 Hierarchische Modelle . . . 15
2.2.7 Ausführung eines SIB-Graphen . . . 16
2.3 jABC-Plugins . . . 18
2.3.1 Plugin-Schnittstelle . . . 18
2.3.2 Tracer . . . 19
2.3.3 Genesys . . . 19
2.3.4 LocalChecker . . . 20
2.3.5 GEAR . . . 20
2.3.6 SIBCreator . . . 21
2.4 Extensible-Markup-Language . . . 21
2.5 Namensräume . . . 24
2.6 XPath 2.0 . . . 25
2.6.1 Datenmodell . . . 25
2.6.2 Sequenzen . . . 27
2.6.3 Pfade . . . 29
2.6.4 Achsen . . . 29
2.6.5 Knotentypen . . . 30
2.6.6 Datentypen . . . 31
2.6.7 Typdeskriptoren für Sequenzen . . . 32
2.6.8 Dokumentreihenfolge . . . 32
2.6.9 Pfadausdrücke . . . 32
2.7 XSLT 2.0 . . . 34
2.7.1 Deklarative Sprache . . . 34
2.7.2 Entwurfsmuster . . . 35

2.7.3 Sequenzkonstruktoren . . . 35
2.7.4 Template-Regeln . . . 36
2.7.5 Benannte Templates und Funktionen . . . 37
2.7.6 Variablen und temporäre Bäume . . . 37
2.7.7 Stylesheet-Module . . . 38
2.7.8 Ausführung einer Transformation . . . 38

3 Lightweight-Declarative-Coordination 41
3.1 Auswahl des Konzepts . . . 42
3.1.1 Koordination in ausführbaren SIB-Graphen . . . 42
3.1.2 Unterteilung in zwei Modellierungsebenen . . . 44
3.1.3 Unterteilung in drei Modellierungsebenen . . . 45
3.2 Ausführung von Transformationen . . . 49
3.3 Tree-Transformer-Graph . . . 50
3.3.1 Verifikationsmethoden . . . 51
3.3.2 Beispiel eines TT-Graphen . . . 51
3.3.3 Tree-Transformer-SIBs . . . 55
3.3.4 Achsen . . . 58
3.3.5 Aufbau von TT-Graphen . . . 59
3.3.6 Zusammenfassung . . . 62
3.4 XML/XSL-Graph . . . 63
3.4.1 Verifikationsmethoden . . . 64
3.4.2 Beispiel eines XSL-Graphen . . . 64
3.4.3 Beispiel eines XSL-SIBs . . . 66
3.4.4 XML-SIBs . . . 67
3.4.5 XSL-SIBs . . . 70
3.4.6 Achsen . . . 71
3.4.7 Zusammenfassung . . . 71
3.5 Fazit . . . 73

4 Realisierung 75
4.1 Systemaufbau . . . 75
4.2 Bootstrapping . . . 76
4.2.1 Bootstrapping-Stufe-1 . . . 78
4.2.2 Bootstrapping-Stufe-2 . . . 79
4.2.3 Integration der einzelnen Schritte . . . 81
4.2.4 Laden von Transformationsgraphen . . . 82
4.2.5 Ausführen von Transformationsgraphen . . . 83
4.2.6 Umwandlung von Transformationsgraphen nach XSLT . . . 84
4.2.7 Zusammenfassung . . . 85
4.3 Generierung von Transformationsgraphen . . . 86
4.4 Import von Stylesheets . . . 87

4.5 Implementierung in Java 88
4.5.1 Anbindung des Plugins an jABC 89
4.5.2 Funktionen zur Unterstützung der Modellierung 89
4.5.3 SIB-Klassen 90
4.5.4 Komponenten für die ausführbaren SIBs 92
4.6 Testmethoden 96

5 Anwendungsbeispiel 99
5.1 XML-Serialisierung 100
5.2 Starttransformation 100
5.3 Modellierung des TT-Graphen 101
5.4 Wiederverwendung von Komponenten 103

6 Ergebnisse und Ausblick 105
6.1 Ergebnisse 105
6.2 Probleme 107
6.3 Ausblick 108

1 Motivation

Die zunehmende globale Vernetzung durch das Internet eröffnet viele neue Möglichkeiten der Kommunikation. Daraus resultiert ein viel stärkerer Datenaustausch zwischen Softwaresystemen. Diese beruhen auf Datenbanksystemen, wie zum Beispiel relationale oder objektrelationale Datenbanken, die diesen Anforderungen nicht alleine gerecht werden. Sie basieren auf der Idee einer zentralen Verwaltung von Daten. Einer Datenbank liegt ein im Voraus festgelegtes Schema zugrunde. Bei vielen Anwendungen müssen hingegen heterogene, verteilte Datensätze verwaltet und ausgetauscht werden, die sich nur schwer in ein festes Datenmodell einfügen lassen.

Semistrukturierte Daten basieren nicht auf einem festen Schema und sind daher flexibel. Sie können verteilt gespeichert und bei sich ändernden Anforderungen angepasst werden. Die textbasierte Extensible Markup Language (XML) ist eine Datenmodellierungssprache, mit der sich semistrukturierte Daten darstellen lassen. Neben ihrer weiten Verbreitung hat sie den Vorteil sowohl maschinen- als auch menschenlesbar zu sein. Weiterhin kann es, als textbasiertes Format, verteilt verwaltet werden. XML kann dokumentzentrierte, semistrukturierte und strukturierte Daten abbilden. Die Sprache ist hierarchisch aufgebaut. Dadurch lässt sie sich auch für die Darstellung von Bäumen verwenden. Für die Modellierung von Datenschemata steht unter anderem die Sprache *XML-Schema* [29] zur Verfügung, die ihrerseits in XML-Syntax notiert wird. Für die Validierung von Dokumenten ist freie Software vorhanden. Anders als die eingeschränkte Verwendungsmöglichkeit von Datenbanken lässt sich XML unter anderem für die Darstellung von Webseiten (XHTML [27]), Vektorgrafiken (SVG [24]), Printmedien (XSL-FO [23]), Beschreibungssprachen für die Orchestrierung von Diensten (BPEL [18]) und dem Ein- und Ausgabeverhalten von Web Services (WSDL [25]) einsetzen. Es gibt Ansätze, Datenbanken mit XML zu modellieren. Als Anfragesprache dient XQuery [30]. Die Performanz ist nicht vergleichbar mit traditionellen Datenbanken. Diese bieten jedoch Import- und Exportschnittstellen für XML an. Auf diese Weise lässt sich die Effizienz von Informationssystemen mit festem Datenschema mit der Flexibilität von XML verbinden.

Ein Format wie XML, das flexibel, portabel und mächtig ist, reicht nicht aus, um die Kommunikation zwischen Systemen zu ermöglichen, sondern bildet lediglich deren Grundlage. Erst die Verwendung von standardisierten und flexiblen Schnittstellen, beziehungsweise Formaten, ermöglichen den einfachen Austausch von Daten. Selbst innerhalb einer Organisation ist es aus den unter-

schiedlichsten Gründen schwierig, ein einheitliches Format für Daten zu haben. Viele Softwareprodukte erweitern in jeder neuen Version die Funktionen. Als Folge muss das Format zur Persistierung und für den Austausch angepasst werden. Für Organisationsgrenzen überschreitende Systeme wird dieses Problem eher größer als kleiner. Die Kommunikation zwischen Systemen, die keine einheitliche Schnittstelle haben, erfordert die Konvertierung der Formate.

Ein Online-Versandhandel arbeitet zum Beispiel mit mehreren Paketdiensten zusammen. Die Paketdienste verwenden unterschiedliche XML-Formate für einen Versandauftrag. Ein Datensatz für einen Auftrag enthält etwa Informationen, wie die Maße und das Gewicht der zu versendenden Artikel sowie die Anschrift, die den Zielort identifiziert. Die Formate können sich in Struktur und geforderten Informationen unterscheiden. So bietet vielleicht nur ein Zusteller einen Expressdienst an. Ein Auftrag für diesen Dienstleister muss entweder als Normal- oder Expressversand deklariert werden. Eine vergleichbare Information wird bei einer Anfrage an einen der anderen Dienstleister nicht benötigt. Eine Transformation des Datensatzes im Format des Versandhändlers in das Format des jeweiligen Paketdienstes ermöglicht eine Anpassung an individuelle Anforderungen.

Nicht nur Transformationen zur Erzeugung von Schnittstellen sind von Interesse. Jede kontextfreie Sprache, wie zum Beispiel XML, Java [19] oder C++ [5] lässt sich mit einem Parser in einen *Parse-Tree* übersetzen. Naheliegende Anwendungen hierfür sind unter anderem Refactoring, Codeanalyse und Generierung von Dokumentation (Javadoc [17], Doxygen [4]). Auch die Realisierung einer Template-Engine für die Generierung von Quelltext ist eine interessante Applikation. Weiterhin sind theoretische Gebiete, wie zum Beispiel Sichten auf Graphen, ein Anwendungsfall. Ferner sind viele weitere Strukturen, wie zum Beispiel Verzeichnisstrukturen, hierarchisch aufgebaut und können durch Baumtransformationen verändert werden.

Diese Arbeit setzt sich mit der graphischen Beschreibung von Transformationen strukturierter und semistrukturierter Daten und infolgedessen mit Baumstrukturen im Besonderen auseinander. Zielsetzung ist die Entwicklung des *Tree-Transformer-Plugins* für Java-Application-Building-Center (jABC [15]). Hierbei handelt es sich um ein Modellierungswerkzeug, das am Lehrstuhl für Programmiersysteme der TU Dortmund entwickelt wird.

jABC unterstützt die graphische Modellierung von Softwaresystemen, in sogenannten SIB-Graphen, die aus Knoten und beschrifteten Kanten bestehen. Die Knoten in einem Graphen können eine beliebig komplexe Aufgabe erfüllen. Die Kapselung von Services unterstützt das Konzept der Serviceorientierung. Beschriftete ausgehende Kanten zeigen auf die möglichen Nachfolger. Eine mögliche Semantik von Graphen des jABC ist die Abbildung eines Kontrollflusses auf eine imperative Sprache, wie zum Beispiel Java oder C++ (ausführbare SIB-Graphen). Der Zustand der Anwendung während der Ausführung des Modells

wird in diesem Fall über einen gemeinsamen Speicher realisiert.

Diese Art der Prozessmodellierung beziehungsweise -koordinierung heißt Lightweight Process Coordination (LPC). Modelle können über alle Phasen der Softwareentwicklung verwendet werden. Auf diese Weise sind Modell und Implementierung über den gesamten Entwicklungsprozess konsistent. Die Verbindung von Komponenten zu Knoten eines Graphen ist sehr flexibel und anpassbar. Ein bestehendes Softwaresystem muss nur geringfügig geändert werden, um die Modellierung mit jABC zu ermöglichen. Während des Bearbeitungsprozesses können durch jABC lokale und globale Bedingungen an den Knoten, beziehungsweise an den Graphen, geprüft werden.

Für die Serialisierung der zu transformierenden Daten wird das Format XML verwendet. Die Wahl fiel unter anderem wegen der weiten Verbreitung, dem breiten Spektrum an Tools in verschiedenen Sprachen und der umfangreichen, frei verfügbaren Dokumentation, auf XML. Neben den bereits genannten Vorteilen, existiert die Transformationssprache *XSL-Transformations* (XSLT), die seit Anfang 2007 in der Version 2.0 den Recommendation-Status erreicht hat. Das Tree-Transformer-Plugin übersetzt Transformationsgraphen in diese Sprache.

XSLT ist eine deklarative, regelbasierte Sprache, die XML-Dokumente transformieren kann. Für die Transformation von Dokumenten stehen freie und kommerzielle XSLT-Prozessoren zur Verfügung.

XML und XSLT sowie weitere Spezifikationen, wie zum Beispiel XML-Schema und SVG sind vom W3C [26] standardisiert. All diese Spezifikationen greifen durch den Einsatz der XML-Syntax nahtlos ineinander. Es ist beispielsweise möglich, mit XSLT weitere XSLT-Dokumente zu verändern. Im weiteren Verlauf der Arbeit wird diese Möglichkeit genutzt, um aus dem Format für die graphische Beschreibung einer Transformation das entsprechende XSLT-Dokument zu generieren.

Das Tree-Transformer-Plugin soll eine Erweiterung des LPC-Konzepts für die Modellierung von Anwendungen auf Transformationen strukturierter Daten realisieren. Die Transformationen werden nicht nur in Kontrollflussgraphen aufgerufen, sondern in SIB-Graphen modelliert. Diese Art der Modellierung soll als Erweiterung oder sogar als Ersatz für eine Entwicklungsumgebung für XSLT dienen und nicht nur als Visualisierung von Stylesheets. Letzteres bieten einige XML- und XSLT-Editoren bereits. Die Modellierungsumgebung soll so aufgebaut werden, dass sie viele Fehler während der Entwicklung unterbindet. Der Entwurfsprozess soll vereinfacht werden, indem das Plugin während der Modellierung Bedingungen auf dem Graphen prüft und Fehler visualisiert. Der Aufruf des XSLT-Prozessors zur Fehlererkennung wird somit in vielen Fällen unnötig. Die graphische Darstellung soll, kombiniert mit den Dokumentationsmöglichkeiten der SIB-Graphen, die jABC bietet, einen Transformationsprozess nachvollziehbar machen und die semantische Lücke zwischen den Wünschen des Auftraggebers und dem Verständnis des Auftragnehmers reduzieren.

Ferner werden an das Tree-Transformer-Plugin die folgenden Hauptanforderungen gestellt:

- Ein Transformationsgraph soll leicht in ausführbaren SIB-Graphen aufgerufen werden können, ohne Kenntnisse von XSLT zu benötigen.
- Die Transformationsgraphen sollen von XSLT abstrahieren und einen einfachen Zugang zur Modellierung von Transformationen bieten, ohne Detailkenntnisse über XSLT zu verlangen.
- Die Graphen sollen möglichst viel von der Sprache XSLT abbilden, um auch komplexe Transformationen beschreiben zu können.

Im Verlauf der Arbeit wird sich zeigen, dass diese Anforderungen strukturelle Auswirkungen auf das Konzept und die Realisierung haben.

Neben der Umsetzung der Anforderungen soll eine Effizienzsteigerung gegenüber dem gängigen Entwurfsprozess für XSLT-Transformationen erreicht werden. Ferner sollen die Vorteile von LPC auch für Transformationen genutzt werden. Dazu zählt die einfache Integration in Applikationen, Webanwendungen oder andere Zielanwendungen über die Generatoren von Genesys. Weiterhin soll der Aufruf einer Transformation, zum Beispiel über die *Java APIs for XML Processing* (JAXP[11]) für den Modellierer transparent sein.

Das Plugin soll mit einem starken Einsatz von jABC und Tree-Transformer realisiert werden. Für die Konzeptvalidierung werden daher unter anderem folgende Aufgaben der Arbeit mit den Funktionen des Plugins umgesetzt:

- Übersetzung der Transformationsgraphen nach XSLT über ein Bootstrapping
- Ausführung der Bootstrapping-Schritte über ausführbare SIB-Graphen
- Generierung von Transformationsgraphen für die Verwendung ohne jABC
- Sammlung statistischer Daten über die Transformationsgraphen, die für die Entwicklung des Plugins erstellt wurden

Die Umsetzung erfolgt als Plugin und wird über die entsprechende Schnittstelle in jABC integriert, daher müssen Restriktionen des Werkzeugs eingehalten werden. Das kann in einigen Fällen zu suboptimalen Lösungen führen, die in Kapitel „Probleme“ auf Seite 107 näher beschrieben werden.

Diese Diplomarbeit dokumentiert die Erfahrungen, die bei der Entwicklung des Tree-Transformer-Plugins gemacht wurden. Sie setzt ein grundlegendes Verständnis der folgenden Konzepte und Technologien voraus:

- Objektorientierung [2] und prozedurale Programmierung

- Serviceorientierung [9]
- Imperative [6] und deklarative [8] Programmierung
- Baum-, beziehungsweise Graphentheorie [16]
- Grammatiken und Parser [20]
- Modelchecking [14]
- XML [22] und HTML [27]
- XML-Schema [29]
- XSLT [13, 31] und XPath [12, 28]

Darüber hinaus erforderliche Kenntnisse werden in Kapitel 2 vermittelt. Kapitel 3 beschreibt das Konzept des Plugins für jABC und Kapitel 4 seine Realisierung. In Kapitel 5 wird ein Anwendungsbeispiel dargestellt. Abschließend werden in Kapitel 6 die Ergebnisse zusammengefasst und bewertet sowie ein Ausblick auf neue Möglichkeiten, die sich durch diese Arbeit eröffnen, gegeben.

2 Grundlagen

Dieses Kapitel beschreibt die verwendeten Technologien und erklärt die theoretischen Grundlagen, die für das Verständnis der Konzepte dieser Arbeit unbedingt erforderlich sind. Für eine vollständige Dokumentation der hier vorgestellten Themen sei auf das Literaturverzeichnis verwiesen.

2.1 Konventionen

Die folgenden Konventionen werden in diesem Dokument verwendet:

Begriff
: Neue Begriffe werden durch Kursivschrift gekennzeichnet.

«`Ausdruck`»
: Pfadausdrücke und Werte, wie Zahlen und Zeichenketten, werden im laufenden Text in «»-Klammern gefasst.

`Name`
: Klassennamen und andere technische Namen, wie Attribute und XPath-Achsen werden durch Schrift mit fester Breite hervorgehoben.

`name()`
: Methodennamen enden auf `()`. Sie beginnen mit einem Kleinbuchstaben. Die Parameter der Methode werden ausgelassen.

`<name>`
: Spitze Klammern zeichnen ein XML-Element aus. XSLT-Elemente werden durch den Präfix `xsl` unterschieden (`<xsl:instruktion>`).

`@Name`
: *Java-Annotations* beginnen mit einem `@`-Zeichen.

2.2 jABC

Java-Application-Building-Center (*jABC*) ist ein Werkzeug zur Modellierung von Prozessen und Workflows in hierarchischen (Fluss-) Graphen. Eine Besonderheit des Ansatzes ist, dass zum Einen ein *Anwendungsexperte* ohne Programmiererfahrung in der Lage ist, Modelle (*SIB-Graphen*) mit jABC zu erstellen.

Zum Anderen sind sie nicht nur deskriptiv, wie viele andere graphische Modellierungsansätze, sondern ausführbar, verifizierbar und kompilierbar.

Die SIB-Graphen werden in der graphischen Benutzeroberfläche auf einem hohen Abstraktionsniveau erstellt. Ein IT-Spezialist (*SIB-Experte*) implementiert die Logik der Knoten (*SIBs*) in den Graphen. Der Implementierer kann sich auf das *jABC-Framework* stützen, das die Kernfunktionalitäten für die Modellierung, die Ausführung von Graphen und die Integration von Plugins zur Verfügung stellt. Alle weiteren Funktionen werden über Plugins realisiert. Auf diese Weise ist das Framework flexibel und erweiterbar. Zu den Basis-Plugins gehören:

Tracer führt SIB-Graphen im jABC aus. Er unterstützt ferner die schrittweise Ausführung.

Genesys generiert ausführbaren Code aus SIB-Graphen.

Local- und Modelchecker (GEAR) prüfen lokale und globale Bedingungen.

SIBCreator erzeugt neue SIBs, die entweder Leerimplementierungen sind oder API-Methoden kapseln.

Auf der Modellierungsebene steht einem Anwendungsexperten die graphische Benutzeroberfläche von jABC zur Verfügung. Hier können die SIBs auf einer Zeichenfläche für Graphen arrangiert werden. Für die Modellierung werden keine Programmierkenntnisse benötigt. Die Trennung von Koordination und Implementierung ermöglicht eine bessere Strukturierung des Entwicklungsprozesses. Die Modularisierung in einzelne Komponenten, der SIBs, ermöglicht die Wiederverwendbarkeit von Funktionalitäten. In Kombination mit den Basisplugins bietet jABC die Möglichkeit, ein konsistentes, anpassbares, verifizierbares und ausführbares Modell eines Anwendungsablaufs auf einem hohen Abstraktionsniveau zu gestalten.

In Abschnitt 2.2.1 wird näher auf den Modellierungsansatz und die Architektur eingegangen. Die Abschnitte 2.2.2 und 2.2.3 beschreiben die Bausteine von SIB-Graphen, wie sie implementiert werden und ein Entwurfsmuster für die Schnittstelle zwischen einem Baustein und seiner Funktionalität. SIB-Graphen, *jABC-Projekte* und hierarchische Modelle werden in den folgenden Abschnitten dargestellt. Abschließend erläutert Abschnitt 2.2.7 die Semantik von *ausführbaren SIB-Graphen*.

2.2.1 Lightweight Process Coordination

SIB-Graphen bestehen aus wiederverwendbaren und parameterisierbaren SIB-Instanzen (*Service-Independent-Building-Blocks*) und gerichteten Kanten mit

ein oder mehreren Bezeichnern (*Branches*). Letztere legen die möglichen Übergänge zwischen den Knoten fest. Eine SIB-Instanz ist ein Exemplar eines SIBs (siehe 2.2.2), der eine beliebig komplexe Funktionalität kapselt. Hierbei kann es sich zum Beispiel um eine Datenbankabfrage, einen Web-Service-Aufruf oder einen aufwändigen Algorithmus handeln. Der SIB-Graph modelliert lediglich die Koordination der Prozesse.

Die daraus resultierende Architektur ist in Abbildung 2.1 graphisch dargestellt. Die SIB-Graphen koordinieren die Komponenten (Services) der Geschäfts-

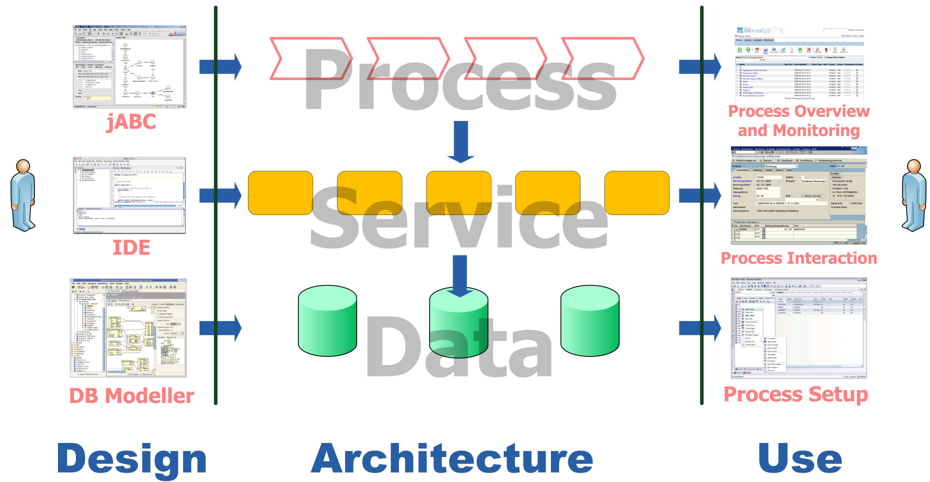

Abbildung 2.1: Architektur des LPC-Konzepts

logik, die der Funktionalität der SIBs entsprechen. Diese Form der Prozessmodellierung und -koordinierung heißt *Lightweight-Process-Coordination* (LPC). Sie hat die nachfolgend beschriebenen Eigenschaften [7].

Einfachheit: Der Prozess konzentriert sich auf Anwendungsexperten, die keine Programmiererfahrung benötigen. Die grundlegende Idee des Modellierungsprozesses ist in der Vergangenheit in praktischen Projekten neuen Teilnehmern innerhalb von weniger als einer Stunde erklärt worden.

Flexibilität: Änderungen an dem Modell sind möglich, ohne die übrigen Teile des Systems zu behindern. Dies stimmt mit dem Prinzip der agilen Softwareentwicklung überein.

Anpassbarkeit: Die LPC-Komponenten, die ein Modell bilden, können frei umbenannt oder umstrukturiert werden, um den Namenskonventionen und Bedürfnissen des Anwendungsexperten zu entsprechen.

Konsistenz: Der Entwicklungsprozess mit jABC verwendet ein konsistentes Modell für die ersten Schritte des Prototyping bis hin zur Fertigstellung des Systems.

Verifikation: Mit Techniken wie dem Local- und Modelchecking wird der LPC-Nutzer unterstützt, während das Modell modifiziert wird. Die Idee ist, lokale und globale Bedingungen an ein Modell zu knüpfen, welche dieses während und vor allem am Ende des Entwicklungsprozesses erfüllen soll.

Serviceorientierung: Bestehende Features oder Anwendungen können einfach in das LPC-Modell integriert werden, indem die Funktionalitäten von einer LPC-Komponente referenziert wird.

Ausführbarkeit: Ein Modell kann verschiedene Ebenen ausführbaren Codes von Beginn der ersten Simulation an, bis hin zur endgültigen Laufzeitimplementierung, enthalten.

Universalität: Das Framework nutzt die plattformunabhängige, objektorientierte Sprache Java als Basis des Systems und kann daher in vielen technischen Bereichen eingesetzt werden. Auch der Zugriff auf andere Programmiersprachen und Architekturen ist möglich.

Die SIB-Graphen repräsentieren eine prozedurale Schicht, deren Abstraktionsniveau frei gewählt werden kann. Sie setzt auf Komponenten auf, die ein SIB-Experte mit einer generischen imperativen Sprache entwickelt. Den Zustand einer Ausführung verwaltet ein gemeinsamer Speicher. Der Modellierer kann das Verhalten einer SIB-Instanz über seine Parameter und Branches beeinflussen. Diese stellt der SIB-Experte als Schnittstelle zur Verfügung (siehe Kapitel 2.2.2).

Die Ausführungsumgebung in jABC stellt das Tracer-Plugin bereit. Ferner übersetzt das Genesys-Plugin SIB-Graphen in eine ausführbare Anwendung.

2.2.2 Service-Independent-Building-Blocks

Der Begriff Service-Independent-Building-Block (SIB) wird für das Konzept verwendet, das die Knoten in SIB-Graphen repräsentieren. Es hat drei Facetten. In Kontexten, in denen die Notwendigkeit besteht, werden sie wie folgt auseinandergehalten:

SIB

Entspricht der konzeptionellen Sicht einer Klasse oder einem Typ von Knoten. Hier wird von der technischen Realisierung als Java-Klasse abstrahiert. Ein SIB ist eine Schablone für Knoten in SIB-Graphen.

SIB-Instanz
Repräsentiert einen Knoten in einem SIB-Graphen. Es handelt sich um eine spezielle Entität eines SIBs. SIB-Instanzen sind Bestandteil der Modellierungsebene.

SIB-Klasse
Bezeichnet die implementierte Java-Klasse. Eine SIB-Klasse ist die technische Realisierung eines SIBs.

Die Bausteine eines ausführbaren SIB-Graphen sind Instanzen von SIBs. Sie repräsentieren eine Komponente, die sie nicht selbst implementieren. Der entsprechende Dienst wird lose an einen SIB gekoppelt (siehe Kapitel 2.2.3). Auf der Modellierungsebene stellt eine SIB-Instanz SIB-Parameter und Branches zur Verfügung. Sie bilden ihre *SIB-Argumente*, mit denen ein Anwender Einfluss auf das Verhalten einer SIB-Instanz nimmt.

SIBs bilden die Schnittstelle zwischen der prozeduralen Sicht von jABC und der objektorientierten Sprache Java. Sie werden als Java-Klasse (SIB-Klasse) realisiert. Eine SIB-Klasse entspricht jedoch nicht dem objektorientierten Programmierparadigma. Im Folgenden werden die Komponenten einer SIB-Klasse und die Unterschiede zur Objektorientierung näher erläutert.

Eine Java-Klasse wird über die *Java-Annotation* `@SIBClass` als SIB-Klasse gekennzeichnet. Die Annotation enthält einen eindeutigen Identifikator (`UID`). Java-Annotation ist eine Sprachfunktionalität von Java, die in der Version 5.0 hinzugekommen ist. Dabei handelt es sich um statische Metainformationen, die zum Beispiel an Klassen-, Methoden-, und Felddeklarationen annotiert werden können.

Ein SIB-Experte kann die Klasse um SIB-Parameter ergänzen, die als globale, nicht statische Felder deklariert werden. jABC unterstützt alle gängigen Java-Typen. Die Felder kann der IT-Spezialist mit `get()`- und `set()`-Methoden erweitern, falls bei einem Lese- oder Schreibvorgang Logik ausgeführt werden soll. Die graphische Oberfläche zeigt die SIB-Parameter an. Ein Anwender kann sie mit den zugehörigen Editoren setzen. Eine SIB-Instanz ist zwar technisch ein Java-Objekt, hat jedoch den Charakter einer Konstanten. Die Implementierung einer SIB-Klasse sollte folglich nicht die Werte eines SIB-Parameters ändern. Dies stellt keine Einschränkung dar, da der Zustand einer Ausführung über einen gemeinsamen Speicher realisiert wird. SIB-Parameter müssen somit keine Zustandsinformationen enthalten.

Die festen Branches eines SIBs werden über ein String-Array gesetzt. Ein zweites String-Array fügt *erweiterbare Branches* (mutable Branches) hinzu. Ist dieses Array deklariert, kann ein Nutzer die darin deklarierten Kantenbezeichner löschen und weitere hinzufügen. Dies ist nur sinnvoll, wenn sie von der Implementierung verwendet werden. Ein Branch ist ein Bezeichner und besteht

lediglich aus einer Zeichenkette. Der Nutzer kann sie in dem jABC-Editor an die ausgehenden Kanten einer SIB-Instanz anfügen. Sie definieren ihre möglichen Nachfolger.

Einige Plugins bieten Funktionalitäten, die eine Implementierung in der Klasse der SIBs benötigen, die sie unterstützen. Zum Beispiel brauchen SIBs in ausführbaren Graphen eine Schnittstelle, über die sie ihre Funktionalität anbinden. Ferner muss für die Verwendung des LocalChecker-Plugins Prüfcode vorhanden sein. Diese Schnittstellen werden durch *Java-Interfaces* realisiert, die die entsprechenden Methoden enthalten. Ein SIB, der LocalChecker-fähig ist, muss beispielsweise das `LocalCheck`-Interface implementieren. Darin ist eine Methode definiert, in der die lokalen Tests umgesetzt werden.

Eine SIB-Klasse kann optional weitere Methoden implementieren, die das Framework auswertet. Eine liefert zum Beispiel ein graphisches Symbol für die Anzeige auf der Zeichenfläche für Graphen, eine weitere bietet Dokumentation zu dem SIB und seinen Argumenten. Die graphische Benutzeroberfläche zeigt die Texte in Tooltips der jeweiligen Komponente als Kontexthilfe an.

Seit jABC-Version-3.6 sind in der jABC-Distribution Standard-SIB-Bibliotheken (SIB-Paletten) enthalten. Sie werden unter dem Oberbegriff *Common-SIBs* zusammengefasst. Sie erfüllen häufig gebrauchte Aufgaben, wie zum Beispiel das Lesen und Schreiben von Dateien auf dem lokalen Dateisystem und Zeichenkettenverarbeitung.

2.2.3 Entwurfsmuster für ausführbare SIBs

Das LPC-Konzept unterstützt die Modellierung vom ersten Entwurf bis zum ausführbaren Programm in dem selben SIB-Graphen. Das erfordert die Trennung der Klasse, die die Funktionalität zur Verfügung stellt und der SIB-Klasse. Auf diese Weise ist es einem Anwendungsexperten möglich, einen Graphen zu erstellen, ohne dass die Komponenten für die Ausführung vorhanden sind. Ein SIB kann auf verschiedene Art und Weise von seiner Funktionalität isoliert werden. Die verschiedenen Varianten heißen *Entwurfsmuster für ausführbare SIBs* (SIB-Pattern). Im Folgenden wird näher auf das *SIB-Adapter*-Entwurfsmuster eingegangen.

Die SIB-Klasse eines ausführbaren SIBs implementiert das Interface `Executable`, das die `trace()`-Methode definiert. Sie wird aufgerufen, wenn die Ausführungsumgebung des Tracers eine SIB-Instanz verarbeitet. Genesys hat ein eigenes Interface für generierbare SIBs, da sich die Implementierungsdetails der generierten Ausführungsumgebung von denen des Tracers unterscheiden. Das Konzept des Entwurfsmusters kann jedoch für beide Umgebungen angewendet werden. Im Folgenden wird das SIB-Adapter-Entwurfsmuster anhand des Tracer-Plugins beschrieben.

Die `trace()`-Methode sollte nicht die Implementierung der Funktionen enthalten, sondern einen Verweis darauf. Das hat zwei Gründe:

1. Es besteht eine starke Kopplung zwischen SIB und Funktionalität.

2. Komponenten von Drittanbietern liegen nicht immer im Quelltext vor. Eine Einbindung in den eigenen Code ist in dem Fall nicht möglich. Ferner ist die Integration von fremdem Progammcode fehleranfällig und wartungsintensiv.

Ein SIB-Adapter ist eine Java-Klasse, die die Funktionalität von ein oder mehreren SIBs verwaltet. Er hat pro SIB eine Methode, die auf die Implementierung verweist. Der SIB-Experte benötigt zur Kompilierzeit demnach alle Abhängigkeiten von Bibliotheken zur Ausführung der SIBs. Während der Modellierung werden nur die SIB-Klasse und der SIB-Adapter benötigt. Falls eine Bibliothek nicht geladen werden kann, fängt die `trace()`-Methode den Fehler ab. Der Anwendungsexperte kann weiterhin modellieren und seine SIB-Graphen ausführen. Lediglich die SIB-Instanzen, deren Adapter eine Abhängigkeit nicht laden kann, sind ohne Funktion. Nach Möglichkeit sollte ein SIB-Adapter nicht von mehreren Bibliotheken abhängen, da alle seine SIBs keinen Zugriff auf ihre Funktionalität haben, wenn eine Bibliothek nicht geladen werden kann. Abbildung 2.2 zeigt

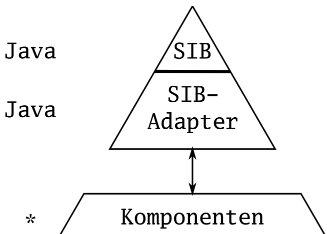

Abbildung 2.2: Das SIB-Adapter-Entwurfsmuster

das SIB-Adapter-Entwurfsmuster graphisch. Die Dreiecksform repräsentiert das Aufwandsverhältnis der Elemente bei der Implementierung. SIBs haben eine sehr schlanke Implementierung, da sie keine eigene Funktionalität besitzen. Der SIB-Adapter ruft die Methoden der Komponenten auf und kann daher komplexer sein. Die oberen beiden Ebenen werden in Java implementiert. Sie werden in der gleichen Laufzeitumgebung ausgeführt. Den größten Teil der Implementierung machen die Dienste aus. Sie sind von dem Adapter getrennt, weil sie weder in Java implemeniert noch auf dem gleichen Rechner ausgeführt werden müssen. Dies wird durch das «*» links neben dem Block für die Komponenten visualisiert. Die Anbindung eines Dienstes an den SIB-Adapter kann über eine beliebige Schnittstelle, wie zum Beispiel *Remote-Method-Invocation* (RMI), *Java-Native-Interface* (JNI), *Corba* oder einen einfachen Methodenaufruf, falls

die Komponente in Java implementiert ist, erfolgen. Die Komponenten der SIBs, die Tree-Transformer verwendet, sind alle in Java implementiert.

Listing 2.1 zeigt den Aufruf einer SIB-Adapter-Methode. Er fängt den `NoClassDefFoundError` ab, der auftritt, falls der Adapter eine Klasse lädt, die nicht gefunden werden kann. In diesem Fall wird die SIB-Instanz über den `error`-Branch verlassen.

```
1  /**
2   * trace()-Methode des Executable-Interfaces.
3   * @param env
4   *  Das ExecutionEnvironment ermöglicht Zugriff
5   *  auf den Kontext.
6   * @return
7   *  Der Branch, über den der Nachfolger des SIBs
8   *  bestimmt werden soll.
9   */
10 public String trace(ExecutionEnvironment env) {
11   // Holt Java-Dokument aus dem Kontext.
12   String document = env.get("java-document");
13   try {
14     // Ruft eine Funktion des SIB-Adapters auf.
15     // Wandelt das Javadokument in XML um.
16     String xml = XMLSIBAdapter.
17       javaToXML(document);
18     // Legt XML-Dokument in den Kontext.
19     env.put("java-xml-document", xml);
20     // Verlässt SIB über "success"-Branch.
21     return "success";
22   }
23   catch(NoClassDefFoundError cnfe) {
24     // Verlässt das SIB über den "error"-Branch.
25     return "error";
26   }
27 }
```

Listing 2.1: Aufruf eines SIB-Adapters

2.2.4 SIB-Graphen

In der jABC-Terminologie heißt ein Modell SIB-Graph. Er besteht aus SIB-Instanzen, benannten Kanten (Branches) sowie Modellparametern und -kanten. Die Argumente eines Modells sind globale SIB-Parameter und -Branches. Diese

können im jABC-Editor an einzelnen Knoten gesetzt werden. Sie ähneln konzeptionell den globalen Variablen einer Klasse. Bei hierarchischen SIB-Graphen bilden diese Eigenschaften die Schnittstelle zu dem übergeordneten Graphen (siehe Kapitel 2.2.6). Die SIB-Instanzen in einem Modell erhalten jeweils einen eindeutigen Identifikator und einen für das Modell eindeutigen Namen, der unterhalb des SIB-Icons angezeigt wird. Die Instanz-ID ist nicht mit der `UID` eines SIBs zu verwechseln. Weiterhin hat jedes Modell eine eindeutige ID, über die es referenziert werden kann und einen beliebigen Namen. Optional kann an einen SIB-Graphen Dokumentation angehängt werden.

2.2.5 Projekte

jABC organisiert SIB-Graphen in Projekten. Ein Projekt entspricht einem Verzeichnis mit einer *Java-Properties*-Datei, die seine Eigenschaften enthält. Es können unter anderem SIB-Pfade gespeichert werden. Sie entsprechen einem *Java-Classpath*-Eintrag an dem nach SIB-Klassen gesucht werden soll. Ein Anwender kann ferner Projektklassenpfadeinträge angeben, um zum Beispiel Java-Bibliotheken für die Funktionalitäten von SIBs einzubinden. Des Weiteren zählt eine optionale Projektdokumentation zu den Eigenschaften.

2.2.6 Hierarchische Modelle

Die Prozesse in jABC können hierarchisch modelliert werden. Für die Kapselung eines SIB-Graphen kann ein Modellierer den Graph-SIB verwenden. Seine SIB-Klasse (`GraphSIB`) ist im Framework enthalten. Eine Instanz von dem `GraphSIB` verweist auf die eindeutige ID eines Modells. Sie kann vom Anwender gesetzt werden. Dieser Vorgang heißt *Programmierung eines Graph-SIB*. Die Schnittstelle zu dem Untergraphen bilden die Modellbranches und -parameter, die zusammengefasst *Modellargumente* genannt werden. Sie sind Argumente von SIB-Instanzen des Untermodells, die der Anwendungsexperte im jABC-Editor veröffentlicht hat. Der Graph-SIB übernimmt sie bei der Programmierung als seine lokalen SIB-Parameter beziehungsweise -Branches.

Mehrere Argumente können zu einem Modellargument verschmolzen werden. Bei SIB-Parametern muss der Typ übereinstimmen. Für Modellargumente kann der Nutzer in dem entsprechenden Editor einen Standardwert setzen. Die Argumente eines Graph-SIBs überschreiben diese Werte.

Das Framework bietet weiterhin die Möglichkeit, fest programmierte Graph-SIBs zu definieren. Sie können nicht über die graphische Oberfläche umprogrammiert werden. Die ID des Modells wird an der SIB-Klasse über die Java-Annotation `@GraphSIB` gesetzt.

2.2.7 Ausführung eines SIB-Graphen

Die folgenden Abschnitte beleuchten die Semantik eines SIB-Graphen von zwei Seiten. Zu Beginn wird abstrakt beschrieben, wie eine Ausführung abläuft. Ferner schildert der zweite Abschnitt die Verarbeitung eines konkreten Beispielgraphen. Hierbei ist unerheblich, ob die Ausführungsumgebung des Tracers ihn verarbeitet oder das Genesys-Plugin eine eigenständige Anwendung generiert hat. Im weiteren Verlauf wird daher allgemein der Begriff Ausführungsumgebung verwendet.

Abstrakte Beschreibung einer Ausführung

Ein ausführbarer SIB-Graph beginnt in einem ausgezeichneten *Start-SIB*. Er wählt, je nach Ausgang seiner Ausführung, einen seiner Branches. Sie beschreiben die möglichen Übergänge zu seinen Nachfolgern. Die jeweilige Ausführungsumgebung bestimmt den entsprechenden Nachfolger und führt ihn aus. Die Ausführung endet, wenn eine SIB-Instanz einen Branch gewählt hat, der nicht existiert, auf keinen Nachfolger zeigt oder als Modellbranch veröffentlicht wurde. In den ersten beiden Fällen erzeugt die Umgebung einen Fehler. Ein Modellbranch kann entweder auf einen in der Graphhierarchie höher liegenden SIB-Graphen zeigen oder, sofern es sich um den Graphen auf der obersten Hierarchieebene handelt, das Ende der Ausführung markieren. Abbildung 2.3 zeigt den Ablauf in einem *deterministischen endlichen Automaten* (DEA).

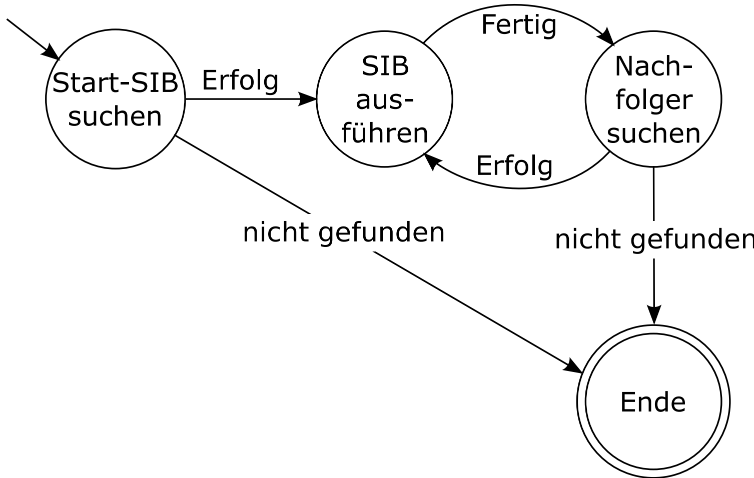

Abbildung 2.3: DEA – Ausführung eines SIB-Graphen

Neben den SIBs gibt es noch Kontroll-SIBs. Sie kapseln keine Funktionalität, sondern manipulieren den Kontrollfluss. Das jABC-Framework liefert beispielsweise den `ForkSIB`, der die Nachfolger in nebenläufige *Threads* aufteilt und

den `JoinSIB`, der Ausführungsstränge wieder zusammenführt. Weiterhin bietet es Kontroll-SIBs für die Vermeidung von *Race-Conditions* bei dem Zugriff auf den gemeinsamen Speicher. Bei derartigen SIBs beschränkt sich das Suchen der Nachfolger nicht zwingend darauf, einem Branch zu folgen. Ein `ForkSIB` hat zum Beispiel mehrere Nachfolger. Ferner wartet ein `JoinSIB` auf mehrere Vorgänger, ähnlich dem *Barrier*-Konzept für nebenläufige Prozesse, bevor es zu einem Nachfolger weiterleitet. Aus diesem Grund ist in Abbildung 2.3 „Nachfolger suchen“ ein eigener Zustand.

Konkrete Ausführung eines Beispielgraphen

Der ausführbare SIB-Graph in Abbildung 2.4 stellt die Ausführung einer Transformation mit dem Tree-Transformer-Plugin dar, das in Kapitel 3 und 4 näher beschrieben wird. Das Modell übersetzt einen Java-Quelltext in einen XML-

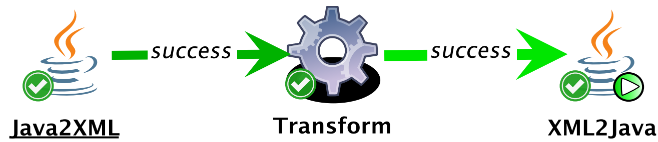

Abbildung 2.4: Ausführung eines konkreten SIB-Graphen im Tracer

Baum und transformiert ihn mit einem Stylesheet. Anschließend wird das Ergebnis in Java-Code konvertiert.

Abbildung 2.4 zeigt den SIB-Graphen während er mit dem Tracer-Plugin schrittweise ausgeführt wird. Die grüne Markierung der Kanten zeigt, wo sich die Ausführung gerade befindet. Je dunkler die grüne Farbe und je dünner die Zeichenstärke der Pfeile, desto mehr Schritte liegt die Verarbeitung dieses Branches zurück. Der Start-SIB wird durch die Unterstreichung des Namens kenntlich gemacht. Ferner veröffentlichen SIB-Instanzen mit fett geschriebenem Namen ein oder mehrere Argumente.

Alle SIB-Instanzen in diesem SIB-Graphen haben als Namen den Klassennamen ihrer SIB-Klasse. Der Graph startet in einem `Java2XML`-SIB, der ein Java-Dokument in eine XML-Repräsentation übersetzt. Das XML-Dokument wird in den gemeinsamen Speicher gelegt. Der nachfolgende SIB ist eine Instanz von dem `Transform`-SIB. Sie entnimmt das Java-Dokument als XML-Baum dem Speicher. Anschließend führt sie mit einem XSLT-Prozessor eine Transformation auf dem XML-Baum aus und legt das Ergebnis in den Speicher.

Die `XML2Java`-SIB-Instanz holt das transformierte XML-Dokument aus dem gemeinsamen Speicher und wandelt es in Java-Quelltext um. Der SIB-Graph wird über den `success`-Modellbranch verlassen (siehe Kapitel 2.2.6).

Tritt in `Java2XML`, `Transform` oder `XML2Java` ein Fehler auf, wird der aktuelle SIB-Graph über einen `error`-Modellbranch beendet.

2.3 jABC-Plugins

jABC besteht aus einem Editor- und einem Framework-Modul. Ersterer bietet eine graphische Oberfläche, mit der ein Anwendungsexperte SIB-Graphen modellieren kann. Des Weiteren verwaltet der Editor die Projekte. Das Framework ist das Back-End-System. Es bietet die Kernfunktionalitäten für die Modellierung und Ausführung von Graphen und eine Schnittstelle für Plugins. Im Folgenden werden einige für die Arbeit wichtige Komponenten der Plugin-Schnittstelle und die Basis-Plugins dargestellt.

2.3.1 Plugin-Schnittstelle

Plugins werden in jABC über eine Klasse identifiziert, die das Java-Interface `Plugin` implementiert. Es enthält die Methoden `start()` und `stop()`, mit der sich ein Plugin an- beziehungsweise abmeldet. Bei der Anmeldung kann es Menüpunkte und Inspektoren in den Editor einbinden. Ein Anwender kann über diese Elemente mit dem Plugin interagieren. Weiterhin können *Java-Listener* an Datenstrukturen des Frameworks angemeldet werden. Eine Klasse, die das Interface `ModelListener` implementiert, kann zum Beispiel bei dem `GraphModelHandler` registriert werden. Dieser Listener wird aufgerufen, wenn ein *Ereignis* (Event) auf dem aktuellen SIB-Graphen aufgetreten ist. Hierzu zählt unter anderem das Hinzufügen und Löschen von SIB-Instanzen oder das Setzen und Entfernen von Branches. Ein Plugin kann auf diese Weise auf Änderungen an dem aktuellen SIB-Graphen reagieren.

Weiterhin können in der `start()`-Methode alle nötigen Vorbereitungen für den Betrieb getroffen werden. Tracer könnte beispielsweise dafür sorgen, dass ein *Thread-Pool* initialisiert wird, der Threads für die Ausführung von SIB-Graphen zur Verfügung stellt. Die `stop()`-Methode wird aufgerufen, bevor das Plugin entladen wird. Es kann hier einen definierten Endzustand herstellen. Tracer könnte zum Beispiel dafür sorgen, dass alle laufenden Ausführungen beendet werden.

Das `Plugin`-Interface hat ferner Methoden, die Informationen über das Plugin zur Verfügung stellen, wie zum Beispiel der Name und das Interface, das SIBs implementieren, die kompatibel zu dem Plugin sind.

Weiterhin kann der Status einer SIB-Instanz bezüglich eines Plugins mit einem kleinen graphischen Symbol angezeigt werden. Der Anwender vergibt die vier Ecken der SIB-Icons an die Plugins seiner Wahl. Das Framework ruft die entsprechende Methode im Plugin für eine SIB-Instanz auf und legt das resultierende Overlay-Icon an der entsprechenden Position über das SIB-Icon.

Neben der Verbindung zu dem Framework gehört auch die Veröffentlichung einer eigenen *API* zu der Schnittstelle eines Plugins. Auf diese Weise können Plugins voneinander profitieren.

2.3.2 Tracer

Tracer erweitert jABC um eine Ausführungsumgebung innerhalb des Editors. Ein Anwendungsexperte kann jederzeit seine SIB-Graphen testen. Zu den Ausführungsmodi zählen unter anderem die kontinuierliche und die schrittweise Ausführung. Bei der zweiten Variante kann der Nutzer in Einzelschritten ein Modell ausführen, wobei ein Overlay-Icon zeigt, welche SIB-Instanz gerade ausgeführt wird. Die besuchten Kanten werden grün markiert (siehe Abbildung 2.4).

Ein SIB-Graph muss die folgenden Voraussetzungen erfüllen, damit der Tracer sie ausführen kann:

- Genau eine SIB-Instanz ist als Start-SIB markiert.
- Alle SIB-Klassen der Knoten implementieren das `Executable`-Interface.

2.3.3 Genesys

SIB-Graphen können nicht nur in der Ausführungsumgebung des Tracer-Plugins ausgeführt werden, das Genesys-Plugin übersetzt sie ferner in ausführbare Anwendungen. So entsteht keine semantische Lücke zwischen der Intention eines Modells und seiner Implementierung. Auch Eigenschaften, die durch LocalChecker oder GEAR an dem Modell geprüft wurden, bleiben erhalten. Ein Anwendungsfall sind die Codegeneratoren von Genesys. Sie sind ebenfalls generierte SIB-Graphen.

Genesys bietet verschiedene Generatoren an. Für diese Arbeit relevant sind der *Java-Class-Extruder* und der *Java-Pure-Generator*. Ersterer generiert eine Anwendung, die – ohne die graphische Oberfläche des Editors – Tracer für die Ausführung verwendet. Folglich hängt der generierte Code von der Framework-Bibliothek und den verwendeten SIB-Klassen ab. Vorteilhaft ist, dass viele der erweiterten Funktionen des Tracer-Plugins unterstützt werden. Hierzu zählen zum Beispiel hierarchische Modelle über den Graph-SIB und Nebenläufigkeit über den `ForkSIB` und `JoinSIB`.

Ein SIB, der den Java-Class-Extruder unterstützt, implementiert lediglich das `CodeGeneratorBlock`-Interface. Es definiert die Methode `execute()`, die in den meisten Fällen an die `trace()`-Methode delegiert werden kann, da sich die Implementierung deckt. Das setzt vorraus, dass der SIB auch das `Executable`-Interface implementiert. Nachteile sind ein hoher Speicheraufwand und eine geringe Performanz, die auf eingeschränkten Systemen problematisch sein können.

Der Code, der mit dem Java-Pure-Generator erzeugt wird, benötigt weder die SIB-Klassen noch das Framework. Er generiert eine Anwendung aus den Funktionalitäten der SIBs. Hierzu ruft er direkt die entsprechenden Methoden in dem SIB-Adapter auf, der in diesem Kontext Service-Adapter heißt. Die SIB-Klassen implementieren das Interface `Generatable`. Von der Methode wird ein Verweis

auf die Methode des Service-Adapters erzeugt. Mit dessen Hilfe generiert Genesys einen eigenen Aufruf. Anwendungen, die mit einem Pure-Generator generiert wurden, laufen schneller und benötigen weniger Speicher als die von Extrudern. Im Gegenzug werden einige Funktionen, wie zum Beispiel Nebenläufigkeit nicht unterstützt. Ferner müssen sie das SIB-Adapter Entwurfsmuster verwenden.

2.3.4 LocalChecker

LocalChecker ist ein Plugin, das lokale Bedingungen an SIB-Instanzen prüft. Hierzu implementiert eine SIB-Klasse das `LocalCheck`-Interface. Es definiert die `checkSIB()`-Methode, die den Prüfcode für die SIB-Klasse enthält. Die einzelnen Tests können Nachrichten erzeugen, die, ähnlich wie bei einem *Logger*, in unterschiedliche Schweregrade unterteilt sind. Die Meldungen können über die Benutzeroberfläche während des Modellierens abgefragt werden.

SIB-Graphen können entweder permanent überwacht oder auf Anfrage geprüft werden. Bei der kontinuierlichen Prüfung wird das Plugin über einen *Modell-Listener* informiert, wenn sich etwas in dem Graphen geändert hat. Dann wird eine Prüfung der beteiligten SIB-Instanzen durchgeführt. Fügt der Nutzer einen SIB ein, wird die entsprechende Instanz getestet. Zieht er eine Kante, sind daran zwei SIB-Instanzen beteiligt. So bleibt der LocalChecker immer auf dem aktuellen Stand. Die Nachricht mit dem höchsten Schweregrad bestimmt den Status einer SIB-Instanz. Er wird über ein Overlay-Icon an dem entsprechenden SIB-Icon markiert.

Ist die Prüfung auf Anfrage aktiviert, initiiert der Modell-Listener keine Tests, sondern ändert das Overlay-Icon der beteiligten SIB-Instanzen auf „Ungetestet“, in Form eines Fragezeichens. Der Anwender kann über die Oberfläche einen Test des gesamten SIB-Graphen starten.

2.3.5 GEAR

GEAR (**G**ame-based, **E**asy **A**nd **Reverse** model-checking) dient der Verifikation von SIB-Graphen in jABC. Die Funktionen basieren auf theoretischen Grundlagen aus dem Bereich des Modelchecking. So wie LocalChecker lokale Bedingungen an SIBs und ihre unmittelbaren Vorgänger und Nachfolger prüft, verifiziert GEAR globale Anforderungen an einen SIB-Graph. Die zu verifizierenden Eigenschaften werden über temporallogische Formeln definiert. Sie stellen eine strukturierte, mathematisch präzise Beschreibung der Anforderungen an einen SIB-Graphen dar. Weiterhin können die Formeln über das FormulaBuilder-Plugin graphisch beschrieben werden.

2.3.6 SIBCreator

SIBCreator erweitert jABC um die Möglichkeit, SIBs zu generieren. Ein Anwender kann entweder Leerimplementierungen erzeugen oder SIBs, die dem SIB-Adapter-Entwurfsmuster folgen und eine beliebige API-Methode kapseln.

Im ersten Fall können folgende Komponenten einer SIB-Klasse automatisch erstellt werden:

- SIB-Parameter
- Branches
- Ein SIB-Icon
- Leerimplementierungen von Plugin-Interfaces
- Dokumentation

Ein Anwendungsexperte kann sie über die graphische Benutzeroberfläche erstellen und in seinen Modellen verwenden. Ein SIB-Experte liefert die Implementierung nach.

Im zweiten Fall wird der Generierungsprozess in zwei Phasen aufgeteilt. In der ersten Phase werden einige API-Methoden zu einem SIB-Adapter zusammengefasst. Diese können über die graphische Benutzeroberfläche ausgewählt werden. Die Methode wird über eine Annotation mit zusätzlichen Informationen über SIB-Parameter, SIB-Icon und Dokumentation der zu generierenden SIB-Klasse angereichert. In der zweiten Phase werden aus dem SIB-Adapter die SIBs generiert. Sie implementieren das `Executable`-, `CodeGeneratorBlock`- und `Generatable`-Interface. Auf diese Weise können sie mit Java-Class-Extruder, Java-Pure-Generator und Tracer verwendet werden.

2.4 Extensible-Markup-Language

Die Extensible-Markup-Language (XML) ist eine textbasierte Sprache zur Beschreibung von hierarchischen Daten. Datenobjekte werden durch XML-Dokumente repräsentiert. Sie haben einen formalen[1] und logischen Aufbau.

Zu dem formalen Aufbau gehört unter anderem eine optionale XML-Deklaration, die Informationen über das Dokument angibt. Hierzu gehört etwa das verwendete *Encoding* und die Version der XML-Spezifikation, die bei dem Parsen des Dokuments berücksichtigt werden soll.

[1] Die XML-Spezifikation [22] verwendet den Ausdruck „physischer Aufbau“. Hier wird von der Standardbenennung abgewichen und der Begriff „formaler Aufbau“ verwendet.

Beispiel 1: XML-Deklaration
Die folgende XML-Deklaration verrät, dass `Latin-1` als Encoding und die Spezifikation 1.0 verwendet wird:

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
```

Listing 2.2: XML-Deklaration

Der logische Aufbau entspricht der Baumstruktur des XML-Dokuments. Die hierarchische Struktur wird durch benannte Elemente aufgebaut. Sie beginnen mit einem benannten Start-Tag `<Elementname>` und werden mit einem End-Tag `</Elementname>` abgeschlossen. Optional kann sich zwischen den Tags Inhalt befinden. Er besteht unter anderem aus Text, Kommentaren und Elementen. Der Inhalt muss abgeschlossen sein bevor das End-Tag des Elements auftritt. Auf diese Weise bleibt die Hierarchieeigenschaft erhalten. Ein leeres Element kann auch zu einem verkürzten Tag `<Elementname />` zusammengefasst werden. Ein Start- beziehungsweise verkürztes Tag kann ferner benannte Attribute enthalten, die aus ihrem Namen beziehungsweise Schlüssel und einem Wert bestehen. Die Attribute eines Elements müssen einen eindeutigen Namen haben.

Ein Dokument, das diese Eigenschaften erfüllt, ist ausbalanciert. XML-Dokumente müssen zudem wohlgeformt sein. Diese Eigenschaft verlangt, dass es genau eine Wurzel besitzt, das Dokumentelement.

Beispiel 2: Eigenschaften von XML-Dokumenten
Das XML-Dokument in Listing 2.3 ist wohlgeformt. Das schließt ein, dass es ausbalanciert ist.

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <order>
3   <article number="23-48-24-32" amount="1"
4     price="200" currency="EUR"/>
5   <article number="23-48-24-33" amount="10"
6     price="20" currency="USD">
7     <note>Bitte zusammen versenden</note>
8   </article>
9 </order>
```

Listing 2.3: Wohlgeformtes XML-Dokument

Das äußerste Element ist das Dokumentelement. Die Zeilen drei und vier enthalten ein leeres, verkürztes Element. In Zeile fünf bis acht befindet sich ein

<article>-Element, das ein <note>-Element beinhaltet. Listing 2.4 zeigt ein ausbalanciertes Dokument. Es hat kein eindeutiges Dokumentelement, sondern mehrere <article>-Elemente auf oberster Ebene. Es ist demnach nicht wohlgeformt und kein XML-Dokument.

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <article number="23-48-24-32" amount="1"
3      price="200" currency="EUR"/>
4  <article number="23-48-24-33" amount="10"
5      price="20"
6    currency="USD">
7    <note>Bitte zusammen versenden</note>
8  </article>
```

Listing 2.4: Ausbalanciertes Dokument

Das schließende Tag des zweiten <article>-Elements in Listing 2.5 steht vor dem End-Tag des enthaltenen <note>-Elements. Die Hierarchieeigenschaft wird hierdurch verletzt. Das Dokument ist folglich weder ausbalanciert noch wohlgeformt.

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <order>
3    <article number="23-48-24-32" amount="1"
4      price="200" currency="EUR"/>
5    <article number="23-48-24-33" amount="10"
6      price="20" currency="USD">
7      <note>Bitte zusammen versenden
8      </article>
9    </note>
10 </order>
```

Listing 2.5: Nicht ausbalanciertes Dokument

Eine weitere Eigenschaft von XML-Dokumenten ist die Validität. Ein Dokument kann geprüft werden, falls es eine *Document-Type-Definition* (DTD), eine Referenz auf ein XML-Schema-Dokument oder eine andere Schema-Sprache enthält. Entspricht das XML-Dokument den definierten Regeln, gilt es als valide. Seit XSLT 2.0 können Pattern für *Template-Regeln* nicht nur auf Übereinstimmung des Namens eines Elements oder Attributs prüfen, sondern auch auf den Typ einer *XML-Schema-Definition*. Die XSLT-2.0-Spezifikation unterteilt entsprechend in *Basic-* und *Schema-Aware*-Prozessoren. Derzeit sind nur Erstere für Java frei verfügbar. Die Transformationen in dieser Arbeit beschränken sich

aus diesem Grund auf die Funktionalitäten eines XSLT-Basis-Prozessors. XML-Schema-Dokumente und DTDs sind somit im weiteren Verlauf von nachrangiger Bedeutung.

2.5 Namensräume

Namensräume (Namespaces) dienen der Vermeidung von Namenskollisionen in XML-Dokumenten. Jedes Element kann beliebig viele Namensraum-Definitionen enthalten, die eine *Uniform-Resource-Identificator* (URI) einem Präfix zuweisen. Ein Sonderfall ist der *Standard-Namensraum* (Default Namespace), der keinem Präfix zugeordnet ist. Eine Deklaration gilt für das aktuelle Element und dessen Attribute sowie alle Nachfolger. Das entspricht dem Teilbaum, der von dem betrachteten Element aufgespannt wird. Ein Namensraum kann somit bereits in dem gleichen Element verwendet werden, in dem es definiert wurde. Elemente, die tiefer in dessen Teilbaum enthalten sind, können Namensräume überschreiben.

Beispiel 3: Namensraum-Deklarationen
Listing 2.6 enthält einige Deklarationen von Namensräumen. Im Dokumentelement werden ein Standard-Namensraum und ein Namensraum für den Präfix `bt` definiert. In Zeile vier wird der Standard-Namensraum überschrieben. Die Elemente in den Zeilen neun bis dreizehn verwenden den Präfix `bt`.

```
1  <order
2    xmlns="http://www.example.org/article"
3    xmlns:bt="http://www.example.org/bill-to">
4    <!-- ... -->
5    <ship-to
6      xmlns="http://www.example.org/ship-to">
7      <name first="Max" last="Muster"/>
8      <street name="Musterstr." number="24"/>
9      <city name="Dortmund" plz="44227"/>
10   </ship-to>
11   <bt:bill-to>
12     <bt:name first="Marina" last="Pattern"/>
13     <bt:street name="Musterstr." number="42"/>
14     <bt:city name="Dortmund" plz="44227"/>
15   </bt:bill-to>
16 </order>
```

Listing 2.6: Namensraum-Deklarationen

2.6 XPath 2.0

Die *XML-Path-Language-2.0* (XPath 2.0) ist eine Sprache, die unter anderem dazu dient, Teile eines XML-Dokuments zu adressieren. Im weiteren Verlauf wird XPath synonym für XPath 2.0 verwendet. Die Sprache kann Daten verarbeiten, die dem *XPath-Datenmodell* (siehe Kapitel 2.6.1) entsprechen. XSLT-Instruktionen verwenden XPath-Ausdrücke vor allem für die Selektion von Knoten oder atomaren Werten, der Formatierung von Werten sowie für das Abfragen von Bedingungen auf dem Eingabedokument. Eine Teilmenge der Sprachelemente von XPath wird für das *Muster* (Pattern) von Template-Regeln verwendet. Es testet, ob eine Regel auf einen Knoten zutrifft. Ferner findet die Sprache Anwendung in anderen Spezifikationen der XML-Familie, wie zum Beispiel XQuery, XPointer und DOM.

XPath wird in dieser Arbeit ausführlich behandelt, da die Sprache und ihr Datenmodell die Grundlage für die Transformationssprache XSLT bilden. Die Graphen, die eine kanonische Abbildung von XSLT-Konstrukten darstellen, sind nah an das Datenmodell von XPath angelehnt. Zum Einen ist diese Darstellung für einen XSLT-Experten vertraut, zum Anderen vereinfacht es die Übersetzung in XSLT. Die nähere Erläuterung der Konzepte und Begrifflichkeiten hinter XPath, dient dem besseren Verständnis der Konzepte in Kapitel 3. Die verschiedenen Knotentypen des Datenmodells und die Relationen zwischen ihnen werden zum Beispiel in XSL-Graphen abgebildet. Für das Verständnis dieser Umsetzung ist es notwendig, die Grundlagen nachvollziehen zu können.

Die folgenden Abschnitte geben einen Einblick in diese Konzepte. Abschließend werden in Abschnitt 2.6.9 einige Beispielausdrücke gezeigt.

2.6.1 Datenmodell

Das XPath-Datenmodell (XDM) repräsentiert XML-Dokumente nicht in ihrer textuellen, serialisierten Form, sondern als eine baumähnliche Struktur. Die Darstellung ähnelt dem *Document-Object-Model* (DOM [21]), definiert jedoch keine Programmierschnittstelle. Das XDM ist keine fest definierte Datenrepräsentation, sondern ein konzeptionelles Modell, das Datenobjekte und ihre Beziehungen untereinander festlegt. Es beschreibt die Sicht von XPath-Ausdrücken auf ein XML-Dokument. Die interne Realisierung der Datenstrukturen ist implementierungsabhängig.

Ein Pfadausdruck hat immer eine unveränderbare Eingabe und eine Ausgabe. Beides entspricht dem Datenmodell. Folglich ist XPath unter dem XDM abgeschlossen. Ferner kann die Ausgabe eines Ausdrucks als Eingabe für einen weiteren Ausdruck oder Teilausdruck verwendet werden. Dies ist die Grundlage für die Kompositionalität der Sprache (siehe Kapitel 3 der XPath 2.0 Spezifikation [28]). Kompositionalität bedeutet, dass Gleiches simultan durch seman-

tisch Gleiches ersetzt werden darf [1]. Daraus resultiert, dass Ausdrücke überall da verwendet werden können, wo ihr Rückgabewert erlaubt ist. Eine formale Definition der Kompositionalität für eine Sprache wie XPath ist aufgrund der vielen syntaktischen Konstrukte sehr aufwändig. In Satz 1 wird die Definition auf arithmetische Ausdrücke beschränkt.

Satz 1: Kompositionalität für arithmetische Ausdrücke in XPath
Seien $t \in \mathbf{PE}$ (Pfadausdrücke), $u, v \in \mathbf{AExp}$ (arithmetische Ausdrücke), $\texttt{\$x} \in \mathbf{Var}$ (Variablen) und $u \equiv v$. Dann gilt:

$$t[u/\texttt{\$x}] \equiv t[v/\texttt{\$x}]$$

Die dreistellige Abbildung $\cdot[\cdot/\cdot] : \mathbf{PE} \times \mathbf{Aexp} \times \mathbf{Var} \longrightarrow \mathbf{PE}$ beschreibt die syntaktische Substitution für arithmetische Ausdrücke. $t[u/\texttt{\$x}]$ bedeutet, dass im Pfadausdruck t die Variable `$x` durch den arithmetischen Ausdruck u syntaktisch ersetzt wird. Weiterhin stellt die Relation $\cdot \equiv \cdot$ die semantische Äquivalenz von Pfadausdrücken dar. Die semantische Äquivalenz impliziert, dass die Ausdrücke unter allen Belegungen der Variablen den gleichen Wert repräsentieren.

So kann in einem Pfadausdruck etwa ein Teilausdruck durch einen anderen Ausdruck ersetzt werden, der den semantisch gleichen Wert besitzt (siehe Beispiel 4).

Beispiel 4: Kompositionalität
Sei $t =_{df}$ «`$x + 1`», $u =_{df}$ «`1`» und $v =_{df}$ «`(-1 + 2)`», dass heißt, es gilt $u \equiv v$. Dann gilt:

$$\begin{array}{rcl}
 & & t[u/\texttt{\$x}] \\
{[t =_{df} «\texttt{\$x + 1}», u =_{df} «\texttt{1}»]} & = & («\texttt{\$x + 1}»)[«\texttt{1}»/\texttt{\$x}] \\
{[\text{Definition von } \cdot[\cdot/\cdot]]} & = & «\texttt{1 + 1}» \\
{[\text{Satz 1 wegen } u \equiv v]} & \equiv & «\texttt{(-1 + 2) + 1}» \\
{[\text{Definition von } \cdot[\cdot/\cdot]]} & = & («\texttt{\$x + 1}»)[«\texttt{(-1 + 2)}»/\texttt{\$x}] \\
{[t =_{df} «\texttt{\$x + 1}», v =_{df} «\texttt{(-1 + 2)}»]} & = & t[v/\texttt{\$x}]
\end{array}$$

Die beiden Ausdrücke «`1 + 1`» und «`(-1 + 2) + 1`» sind semantisch äquivalent, obwohl eine syntaktische Substitution vorgenommen wurde.

XPath kann weder seine Eingabe verändern, noch Variablen deklarieren, die über den Ausdruck hinaus sichtbar sind. Die einzigen Variablen, die in einem

XPath-Ausdruck erstellt werden, sind Kontextvariablen in `for`-Schleifen und ähnlichen Konstrukten, die nur innerhalb der Kontrollstruktur gültig sind. So bleiben Pfadausdrücke seiteneffektfrei. Diese Eigenschaft ist essentiell für XSLT, da sie auf XPath basiert und ebenfalls keine Seiteneffekte haben soll.

Das Datenmodell ist nicht so streng, wie die XML-Spezifikation. Es akzeptiert als Eingabe auch ausbalancierte Dokumente, die nicht wohlgeformt und folglich keine XML-Dokumente sind. Die Eingabe eines XPath-Ausdrucks wird mit Hilfe des *Evaluation-Context* übergeben. Er dient als Schnittstelle zu einer Host-Sprache, wie zum Beispiel XSLT. Der Kontext beinhaltet etwa die Variablen, die für einen Pfadausdruck sichtbar sind und das Kontext-Item, das die derzeitige Position in der Eingabe beschreibt.

2.6.2 Sequenzen

In vielen objektorientierten Sprachen sind (fast) alle Strukturen Objekte. In Java erbt etwa jede Klasse von `java.lang.Object`. Lediglich primitive Typen bilden eine Ausnahme.

XPath basiert ebenfalls auf einer grundlegenden Struktur, der *Sequenz* (Sequence). Hierbei handelt es sich um eine geordnete Menge von keinem Item, einem Item oder mehreren Items. Alle Datenobjekte werden als solche definiert. Für die Elemente wird, um Mehrdeutigkeiten zu verhindern, der Begriff „Item“ verwendet. Das entspricht auch der Namensgebung in der Spezifikation. Mit „Element“ werden bereits XML-Elemente in XML-Dokumenten und die entsprechenden Knotentypen im XDM bezeichnet. Ein Item kann ein Knoten – dazu zählen unter anderem auch Elementknoten – oder ein atomarer Wert sein und ist selbst eine Sequenz.

Ein Baum oder Teilbaum wird als einelementige Sequenz, die aus dem Wurzelknoten des Baums besteht, definiert. Das XDM ist ein abstraktes Modell, daher muss keines dieser Konstrukte je implementiert werden. XPath erwartet jedoch, dass sich jede Eingabe wie eine Sequenz verhält. Abbildung 2.5 zeigt die Abhängigkeiten zwischen den Komponenten in einem statischen UML-Diagramm.

Sequenzen werden nur über ihre Items definiert. Haben zwei die selben Items in der gleichen Reihenfolge, so sind sie ununterscheidbar. Folglich gibt es auch nur eine eindeutige leere Sequenz, die der Abwesenheit von Daten entspricht.

Sequenzen haben immer eine feste Reihenfolge, auch wenn keine Ordnung festgelegt sein muss. So resultiert der mehrfache Zugriff auf die gleiche Position in dem selben Item. Das erste Item wird mit der Position «`1`» adressiert. Viele Programmierer werden das als ungewohnt empfinden, da das erste Element einer *Liste* im Normalfall mit «`0`» angesprochen wird. Weiterhin können sie gleichzeitig atomare Werte und Knoten enthalten. Die Schachtelung ist nicht möglich, da sich auf diese Weise Baumstrukturen erzeugen ließen, die bereits durch XML-Bäume abgedeckt werden. XPath kann zwar keine XML-Knoten erzeugen, es ist

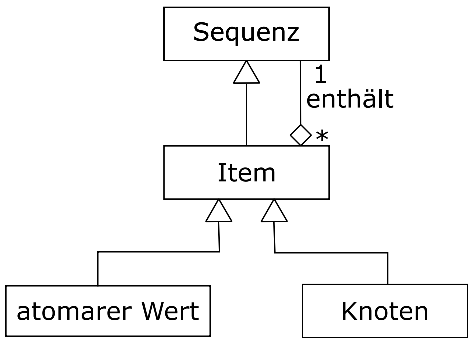

Abbildung 2.5: Abhängigkeiten zwischen den Strukturen im XPath Datenmodell

jedoch möglich, in XSLT welche zu erstellen und über den Evaluation-Context, etwa eine Variable, einem Pfadausdruck zugänglich zu machen. Ferner sollen Sequenzen vor allem XML-Schema-Listentypen darstellen, die ebenfalls nicht geschachtelt werden können.

In der objektorientierten Programmiersprache Java zeigt eine Variable, die mit einem primitiven Typen deklariert und initialisiert wurde, auf den Wert, während Variablen für Objekte eine Referenz auf das Objekt halten. Bei einer Parameterübergabe führt dies zu einer Unterscheidung in *by value*- und *by reference*-Übergabe. Ein Objekt kann von mehreren Variablen referenziert werden, ein Wert eines primitiven Typs muss hingegen bei jeder Übergabe kopiert werden. Im XDM werden atomare Werte wie primitive Typen behandelt. Knoten werden demgegenüber referenziert und können in mehreren Sequenzen oder mehrmals in einer Sequenz enthalten sein.

Die Operationen auf Sequenzen bilden die *Sequenzausdrücke* (Sequence-Expression). Folgende Auflistung erläutert einige Beispiele:

- «`$x, $y`» fasst zwei Sequenzen, in angegebener Reihenfolge, zu einer zusammen. Die Reihenfolge innerhalb der Sequenzen `$x` und `$y` bleibt erhalten.

- «`$x[number]`» adressiert ein Item.

- «`$x[predicate]`» filtert eine Teilsequenz von Items, die das Prädikat erfüllen.

- «`for $y in $x return f($y)`» iteriert über `$x` und führt Funktion `f()` für jedes Item aus. Die Funktion kann ein beliebiger Ausdruck sein. Die Ergebnisse der Funktionsaufrufe werden zu einer neuen Sequenz konkateniert.

- «`$x union $y`» beziehungsweise «`$x | $y`» vereinigt zwei Sequenzen. Ferner können der Schnitt und die Differenz gebildet werden. Hierbei

handelt es sich um Operationen auf Mengen. Sie werden mit Hilfe von Sequenzen simuliert, indem die Ausgabe Duplikate löscht und die Reihenfolge ignoriert.

2.6.3 Pfade

Die Hauptaufgabe von XPath ist das Adressieren von Komponenten eines XML-Dokuments. Die Selektion erfolgt über Pfade, die in einzelne *Schritte* (Steps) unterteilt sind. In einem Schritt wird eine Sequenz von Knoten selektiert. Im Falle von relativen Pfaden beginnt der erste Schritt mit dem Kontextknoten, der über den Evaluation Context durch die Host-Sprache dem Ausdruck verfügbar gemacht wird. Absolute Pfade starten mit dem Dokumentknoten.

Die Auswahl der Knoten in einem Schritt erfolgt über sogenannte *Achsen* (Axis). Sie definieren die Beziehungen zwischen Knoten. Für jeden Knoten `x` des vorhergehenden Schrittes werden alle Knoten gewählt, die mit `x` über die gewählte Achse in Relation stehen. Ein *Knotentest* (Node Test) filtert Knoten über eine Restriktion des Namens oder des Typs. Die Einschränkung auf einen Typen kann sich sowohl auf den Knotentyp als auch auf den Schematyp beziehen. Ferner können optional ein Prädikat oder mehrere Prädikate definiert werden. Nur Knoten, die die Bedingungen erfüllen, werden in die Auswahl eines Schrittes aufgenommen.

Beispiel 5: Pfadausdruck
Folgender Ausdruck beschreibt einen absoluten Pfad, bestehend aus zwei Schritten:

```
1 /order/article[@amount gt 1][@currency eq 'USD']
```

Listing 2.7: Pfadausdruck mit zwei Schritten

Im ersten wird über die `child`-Achse ausgehend vom Dokumentknoten das `<order>`-Dokumentelement selektiert. Der zweite Schritt wählt alle `<article>`-Kindelemente, deren `amount`-Attribut einen numerischen Wert größer als «`1`» und deren `currency`-Attribut den `String`-Wert «`USD`» haben.

2.6.4 Achsen

Eine Achse definiert, für einen Knoten `x` einen Pfad, beginnend in `x`. Er wählt alle Knoten, die zu `x` in der jeweiligen Relation stehen. Der Pfad kann im Baum vorwärts- oder rückwärtsgerichtet sein. Die Knoten werden als Sequenz zurückgegeben. Ist eine Achse für den Knotentyp von `x` nicht definiert, wird die leere Sequenz ausgegeben.

Einige Achsen stehen für Pfade der Länge eins. Sie sind vergleichbar mit den Relationen zwischen Knoten, die im DOM definiert sind. Die `child`-, `parent`- und `attribute`-Achse wählt zum Beispiel alle direkten Kind-, Eltern- beziehungsweise Attributknoten des Kontextknotens. Weitere Achsen definieren längere Pfade. Beispielhaft seien hier die `ancestor`- und `descendant`-Achse angeführt, die alle Nachfolger oder Vorgänger selektieren. Insgesamt definiert XPath dreizehn Achsen.

2.6.5 Knotentypen

Das Datenmodell von XPath definiert sieben Knotentypen, die den Komponenten des Quelldokuments entsprechen. Die folgenden Abschnitte geben eine kurze Übersicht über die wichtigsten Knotentypen und beschreiben die direkten Relationen, die für sie definiert sind. Die Achsen für längere Pfade eines Knotentypen ergeben sich aus den direkten Beziehungen. So ist für alle Knotentypen, die die `child`-Achse haben (Element- und Dokumentknoten) auch unter anderem die `descendant`-Achse definiert. Kommentarknoten und Knoten für Processing-Instructions werden nicht näher beschrieben.

Dokumentknoten

Jedes Dokument hat eine eindeutige Wurzel, den Dokumentknoten. Er ist nicht zu verwechseln mit dem äußersten Element in einem wohlgeformten XML-Dokument, dem Dokumentelement. Das XDM akzeptiert auch ausbalancierte Dokumente und der Dokumentknoten sichert in diesem Fall die Baumeigenschaft. Der Dokumentknoten hat als einziger keinen Elternknoten.

Elementknoten

Elementknoten bilden die Wurzel des Teilbaums, der durch den Inhalt zwischen dem Start- und End-Tag des Elements aufgespannt wird. Der Name entspricht dem Elementnamen. Sie können Kinder, Attribute und Namensräume haben. Jedes Element hat genau einen Elternknoten, der entweder ein Elementknoten oder Dokumentknoten ist.

Textknoten

Ein Textknoten referenziert eine Zeichenkette, die Teil des Inhalts eines Elements ist. Für einen Textknoten ist nur die `parent`-Achse definiert, die bei wohlgeformten Dokumenten immer auf einen Elementknoten zeigt, ansonsten unter Umständen auf einen Dokumentknoten. Textknoten haben keinen Namen.

Attributknoten

Attributknoten sind Kinder von Elementknoten, werden aber über die `attribute`-Achse angebunden, da sie nicht Teil des Inhalts eines Elements sind. Die Reihenfolge der Attribute hat keine Relevanz. Sie bestehen aus einem Namen und einem Wert.

Namensraumknoten

Die `namespace`-Achse verbindet Namensraumknoten mit Element- und Attributknoten. Sie werden von einem Element, das einen Namensraum deklariert, für alle nachfolgenden Knoten als Kopie angehängt, für die der Namensraum definiert ist.

2.6.6 Datentypen

XSLT-Basis-Prozessoren können die Standardtypen der Sprache XML-Schema für atomare Werte verarbeiten. *Komplexe Typen* (Complex Types) und selbst definierte *einfache Typen* (Simple Types) werden nicht unterstützt. Typinformationen können für die Definition von globalen Parametern, Parametern für Template-Regeln oder Funktionen und ihre Rückgabewerte genutzt werden. Des Weiteren werden sie in Pfadausdrücken in *Typtest*-Operatoren (Type Checking), wie «**`instance of`**» und «**`treat as`**», verwendet. Die Standardtypen beinhalten aus Programmiersprachen bekannte Typen wie zum Beispiel `Integer`, `Double`, `String` und `Boolean`. Ferner gehören einige XML-spezifische Typen dazu, wie zum Beispiel die Folgenden:

QName: Ein `QName` enthält einen qualifizierten XML-Namen. Es gibt zwei Darstellungsformen. Lexikalisch besteht er aus einem lokalen Namen `name` oder aus einem lokalen Namen qualifiziert mit einem Präfix `prefix:name`. Intern wird er über den lokalen Namen und eine Namensraum-URI identifiziert. Zwei `QName`-Werte zeigen demnach auch auf den gleichen Namensraum, wenn sie sich in ihrem Präfix unterscheiden. XPath bietet Funktionen, um die einzelnen Komponenten eines Wertes zu erhalten und darauf Bedingungen zu testen.

Der Ausdruck in Listing 2.8 prüft etwa, ob der Kontextknoten den lokalen Namen `article` hat und dem Namensraum «`http://www.example.org/ns`» zugeordnet ist.

```
1 node-name(.) = expanded-QName(
2     "http://www.example.org/ns", "article")
```

Listing 2.8: expanded-QName()

anyURI: Dieser Typ ist eine *Restriktion* (Restriction) auf `String` und beschreibt Zeichenketten, die der Spezifikation einer URI entsprechen.

2.6.7 Typdeskriptoren für Sequenzen

Atomare Typen können über einen `QName` referenziert werden. Für Knoten oder Sequenzen ist mehr notwendig. *Typdeskriptoren für Sequenzen* (Sequence-Type-Constructors) bieten eine Möglichkeit, Typen für Sequenzen mit Knoten und atomaren Werten zu beschreiben. XPath verwendet dieses Konstrukt lediglich in *Typ-Test-Ausdrücken* (Type-Expressions), wie «`instance of`». XSLT setzt sie unter anderem für die Deklaration von Variablen- und Parametertypen ein. Typdeskriptoren sind auch für Basis-XSLT-Prozessoren von Interesse, da nicht nur auf Schemadeklarationen getestet werden kann, sondern auf den Knotentyp, den Namen und die Anzahl der Items.

Im Folgenden sind einige Beispiele von Typdeskriptoren aufgelistet:

- Der Pfadausdruck «`item()*`» akzeptiert jede Sequenz. Das «`*`» legt fest, mit welcher Häufigkeit Items auftreten dürfen. Ein «`*`» steht für kein Item, ein Item oder mehrere Items, «`+`» für ein Item oder mehrere Items und «`?`» für kein Item oder ein Item. Der Ausdruck «`item()`» prüft auf den allgemeinsten Typ der sowohl atomare als auch Knotentypen erlaubt.
- «`xs:string?`» erwartet einen optionalen `String`.
- «`element()+`» nimmt ein oder mehrere Elementknoten an.

2.6.8 Dokumentreihenfolge

Die *Dokumentreihenfolge* (Document Order) ist die Sortierung von Knoten, die innerhalb eines Quelldokuments vorhanden ist. Viele Operationen, wie die Vereinigung, resultieren in Sequenzen ohne Duplikate in Dokumentreihenfolge. Grund hierfür ist, dass Vereinigung, Schnitt und Differenz Mengenoperationen sind. XPath hat jedoch keine Datenstruktur, die einer Menge entspricht. Folglich wird als Sortierung die Dokumentreihenfolge verwendet. Wurde eine der Sequenzen zuvor sortiert, wird diese Abfolge aufgehoben.

2.6.9 Pfadausdrücke

Pfadausdrücke werden in den SIB-Graphen dieser Arbeit nicht explizit modelliert. Demzufolge wird hier nicht näher auf die Syntax der Sprache eingegangen. Lediglich die Konzepte der Spezifikation, die für die Sprache XSLT von Bedeutung sind und Einfluss auf die Konzepte in Kapitel 4 haben, wurden beschrieben.

Für das bessere Verständnis von Pfadausdrücken sind in diesem Abschnitt zusätzlich einige Beispiele mit einer kurzen Beschreibung der jeweils erzeugten Ausgabe aufgeführt:

- «`sum(/order/article/@amount)`» summiert den numerischen Wert der `amount`-Attribute von allen `<article>`-Elementen, die Kind des `<order>`-Dokumentelements sind. Angewendet auf das XML-Dokument in Listing 2.3 auf Seite 22, wäre das Ergebnis «`11`». Die Ausgabe ist unabhängig vom Kontext-Item, da es sich um einen absoluten Pfad handelt. In diesem Beispiel gibt der Ausdruck die komplette Bestellmenge über alle bestellten Artikel zurück.

- Der Ausdruck in Listing 2.9 selektiert alle `price`-Attribute von `<article>`-Elementen, die Kind des Kontextknotens sind und iteriert über diese Sequenz. In der Schleife wird anhand des `currency`-Attributs bestimmt, ob als Währung US-Dollar (USD) oder Euro (EUR) angegeben ist. Es wird davon ausgegangen, dass nur diese Währungen erlaubt sind. Handelt es sich um US-Dollar, wird der Wert in Euro umgerechnet. Weiterhin wird der Preis mit der Bestellmenge des Artikels multipliziert. «`200, 139.28`» ist die Ausgabe des Ausdrucks für das XML-Dokument in Listing 2.3 mit dem Dokumentelement als Kontext-Item und einem Umrechnungskurs (`$USDtoEUR`) von «`0,6964`».

```
1 for $x in article/@price return
2   if ($x/../@currency eq 'USD') then
3     ($USDtoEUR * $x * $x/../@amount)
4   else
5     ($x * $x/../@amount)
```

Listing 2.9: Pfadausdruck

- Der Pfadausdruck in Listing 2.10 bildet den mittleren Wert über alle `price`-Attribute im Dokument. Preise in US-Dollar werden in Euro umgerechnet. Für das Beispieldokument in Listing 2.3 liefert der Ausdruck «`106.964`».

```
1 avg(for $x in //@price return
2   $x *
3   (if($x[../@currency eq 'USD']) then
4     $USDtoEUR
5   else
6     1.0))
```

Listing 2.10: Pfadausdruck

2.7 XSLT 2.0

Die Sprache *XML-Stylesheet-Language-Transformations 2.0* (XSLT 2.0) dient der Beschreibung von Transformationen auf XML-Dokumenten. Syntaktisch handelt es sich bei einer XSL-Transformation um ein oder mehrere Stylesheet-Module, die jeweils ein valides XML-Dokument bezüglich der Schemadefinition der XSLT-Spezifikation ist. Eine Transformation verarbeitet ein Quelldokument oder mehrere Quelldokumente und erzeugt wiederum ein oder mehrere XML-Dokumente. Die Ausführung erfolgt über XSLT-Prozessoren, die ein Stylesheet interpretieren. Die nächsten Kapitel erläutern einige Schlüsselkonzepte der Sprache.

2.7.1 Deklarative Sprache

XSLT ist deklarativ, das heißt, eine Transformation beschreibt ihren Effekt, nicht wie er erreicht wird. Ein Programm in einer imperativen Sprache stellt den Kontrollfluss beziehungsweise den Ablauf der Ausführung dar. Deklarative Sprachen hingegen legen nicht die Reihenfolge der Ausführung fest, sondern beschreiben, was erreicht werden soll. Der deklarative Charakter von XSLT zeigt sich in mehreren Ausprägungen der Sprache:

- Alle *Instruktionen* (Instructions) sind seiteneffektfrei. Die Ausgabe hängt lediglich von der Eingabe ab und ist somit nicht zustandbehaftet. Diese Eigenschaft gilt auch für eine Transformation als Ganzes. Die mehrfache Ausführung auf der gleichen Eingabe führt zu dem gleichen Ergebnis.

- Eine XSLT-Instruktion verändert nicht die Eingabe. Wird ein Teil der Eingabe in die Ausgabe übernommen, werden die entsprechenden Items kopiert.

- Variablen und Parameter können nur einmal mit einem Wert initialisiert werden. Es gibt keinen Zuweisungsoperator, der den gesetzten Wert einer Variablen verändert. Eine Funktion kann zwar mehrfach mit unterschiedlichen Werten für einen Parameter aufgerufen werden, die *Sichtbarkeit* (Scope) des Parameters ist jedoch auf die Ausführung der Funktion beschränkt. Es handelt sich demnach um zwei verschiedene Instanzen der Variablen.

- Die Reihenfolge, in der Instruktionen ausgeführt werden, ist nicht festgelegt. Dies ist nur möglich, da Stylesheets nicht zustandsbehaftet sind, die Eingabe nicht verändert wird und Variablen ihren einmal gesetzten Wert nicht mehr ändern. Ein XSLT-Prozessor kann den Ablauf einer Transformation optimieren. Er muss lediglich sicherstellen, dass er die Ausgabe in der richtigen Reihenfolge erzeugt.

2.7.2 Entwurfsmuster

In dem Standardwerk *XSLT 2.0 3rd Edition Programmer's Reference* [13] unterscheidet der Autor zwischen verschiedenen Entwurfsmustern für Transformationen. Sie können vermischt werden und dienen der Klassifizierung der verschiedenen Herangehensweisen bei der Entwicklung von Stylesheets. Die Wichtigsten werden im Folgenden erläutert.

Das *navigational Design-Pattern* wird vor allem für starre und vorhersagbare Eingabedokumente verwendet und ist ausgabeorientiert. Die Ausführung findet in der Template-Regel für den Dokumentknoten des Eingabedokuments statt. Die Transformation delegiert die Verarbeitung des Inhalts nicht an weitere Regeln. Ergebniselemente erzeugen das Layout der Ausgabe, während Instruktionen die Lücken zwischen ihnen mit Werten aus dem Quelldokument füllen. navigational Stylesheets verarbeiten ein Dokument prozedural. Sie verwenden Variablen für mehrfach gebrauchte Werte. Benannte Templates und Funktionen setzen sie für wiederkehrende Aufgaben ein.

XSLT-Skripte, die dem *regelbasierten Entwurfsmuster* (Rule-Based Design-Pattern) folgen, werden vor allem für XML-Dokumente verwendet, die ein sehr flexibles Format haben. Sie deklarieren Regeln für die Verarbeitung von Knoten, die in der Eingabe vorkommen können. Dabei spielt die Reihenfolge im Quelldokument keine Rolle. Es wird wenig über die Struktur des Eingabe- oder Ausgabedokuments angenommen. Eine Template-Regel delegiert die Verarbeitung von Nachfolgern des Kontextknotens an weitere Regeln. Sie sind untereinander schwach gekoppelt, da eine Template-Regel nicht weiß, welche Regeln für die Nachfolgeknoten ausgewählt werden. Folglich lassen sich regelbasierte Stylesheets durch den Import von Stylesheet-Modulen einfach erweitern.

Das *algorithmische Entwurfsmuster* (Computational Design-Pattern) wird für komplexe Aufgaben verwendet. Sollen Knoten erzeugt werden, die nicht direkt mit Knoten der Eingabe korrespondieren, reichen navigational und regelbasierte Stylesheets nicht aus. Als algorithmische Stylesheets werden solche bezeichnet, die sich in keine der anderen Entwurfsmuster einordnen lassen. Sie zeichnen sich im Normalfall durch eine starke Transformation beziehungsweise Veränderung des Eingabedokuments aus.

2.7.3 Sequenzkonstruktoren

XSLT basiert auf dem XPath Datenmodell. Die Rückgabe einer XSLT-Instruktion ist dementsprechend immer eine Sequenz. Sie wird durch den *Sequenzkonstruktor* (Sequence Constructor) definiert. Eine Ausnahme bilden der Befehl `<xsl:sequence>` und Aufrufe von bennanten Templates und Regeln. Das `select`-Attribut von `<xsl:sequence>` ist ein XPath-Ausdruck. Die Rückgabesequenz des Ausdrucks bildet das Ergebnis der Instruktion.

Aufrufe von benannten Templates oder Template-Regeln geben die Sequenz des zugehörigen Konstruktors zurück.

Ein Sequenzkonstruktor setzt sich aus einer Folge von Elementen zusammen, die Teil des Inhalts der Instruktion sind. Sie lassen sich in zwei Kategorien unterteilen, den *Ergebniselementen* (Literal-Result-Elements) und den Instruktionen. Erstere können Knoten erzeugen, die nicht Teil der Eingabe sind. Ein Ergebniselement ist ein XML-Element, das direkt in die Ausgabesequenz des Sequenzkonstruktors kopiert wird. Letztere haben wiederum eine Sequenz als Rückgabe. Dieser Wert wird evaluiert und in die Rückgabesequenz des Konstruktors aufgenommen.

2.7.4 Template-Regeln

XSLT ist eine regelbasierte Sprache. Regeln werden durch *Template-Regeln* (Template-Rules) repräsentiert. Diese gehören zu den *Top-Level-Deklarationen*. Das bedeutet, sie sind Kindelemente des Dokumentelements eines Stylesheets.

Eine Regel definiert ein Muster, das beschreibt, welche Knoten sie verarbeiten kann. Ferner kann ein optionaler Modus angegeben werden. Ein XSLT-Prozessor befindet sich immer in einem Modus und nur Regeln, die für diesen definiert sind, können für einen Knoten aufgerufen werden. Er ist vergleichbar mit dem Zustand in einem lexikalischen Scanner. Im Standardfall startet eine Transformation im unbenannten Modus (*Default-Modus*) mit dem initialen Knoten des ersten Quelldokuments. Dabei handelt es sich um den entsprechenden Dokumentknoten. Sollten zwei oder mehr Regeln in dem richtigen Modus für einen Knoten zuständig sein, entscheidet die *Priorität* (Priority). Sie kann explizit angegeben werden. Ist dies nicht der Fall, berechnet der XSLT-Prozessor sie aus dem Muster. Haben zwei konkurrierende Regeln die gleiche Priorität erzeugt er einen Fehler und die Transformation bricht ab. Ist keine Regel für einen Knoten definiert, ruft er eine *integrierte Template-Regel* (Built-In Template Rule) auf. Für jeden Knotentyp existiert eine Regel. Textknoten werden zum Beispiel in die Ausgabe kopiert und für Elementknoten wird `<xsl:apply-templates>` aufgerufen, um die Kinder des Elements in dem aktuellen Modus zu verarbeiten. Das bedeutet Elementknoten werden von den Built-In-Regeln unterdrückt.

Der Inhalt einer Template-Regel besteht neben Parameterdeklarationen noch aus einem Sequenzkonstruktor. Er beschreibt die Ausgabe, die für diesen Knoten und seinen Inhalt erzeugt wird. Bei Elementknoten zählt der Teilbaum, der von ihm aufgespannt wird, dazu. Für den Dokumentknoten gibt die verarbeitende Regel folglich das XML-Dokument zurück, das die Transformation für das Eingabedokument erzeugt. Der Vollständigkeit halber sei erwähnt, dass mit der Instruktion `<xsl:result-document>` mehrere Ausgabedokumente erzeugt werden können. Wird kein Gebrauch von diesem Befehl gemacht, erzeugt die Regel, die den Dokumentknoten verarbeitet, das Ausgabedokument.

Mit der `<xsl:apply-templates>`-Instruktion können die Nachfolger eines Knotens verarbeitet werden. Sie ruft Regeln für alle Knoten auf, die in dem Pfadausdruck des `select`-Attributs ausgewählt wurden. Das `modus`-Attribut gibt an, in welchem Modus die Regeln gesucht werden. Ohne Angabe wird der Default-Modus verwendet. Der Wert «`#current`» gibt an, dass der aktuelle Modus beibehalten werden soll.

Der aufrufende Sequenzkonstruktor integriert die Ausgabesequenzen der Regeln in der Reihenfolge, wie sie der `select`-Ausdruck angibt, in seine Ausgabesequenz. Dieses Vorgehen entspricht dem regelbasierten Entwurfsmuster für Transformationen. Die `<xsl:apply-templates>`-Instruktion kann auch Knoten selektieren, die nicht Teil des Inhalts des Kontextknotens sind. Die Knoten müssen nicht einmal zu dem gleichen Dokument gehören. Diese Vorgehensweise ist legitim, folgt jedoch eher dem algorithmischen Entwurfsmuster.

2.7.5 Benannte Templates und Funktionen

Zu den Top-Level-Deklarationen zählen auch die *benannten Templates* (Named Template) und die *Funktionen* (Function). Erstere werden mit `<xsl:call-template>` aufgerufen und haben statt einem Muster einen Namen. Alternativ können Funktionen ihren Platz einnehmen. Sie werden in Pfadausdrücken referenziert. Funktionssignaturen beeinhalten im Gegensatz zu Templates keine benannten Parameter. Die Namen werden zwar für die lokalen Variablen in der Funktion verwendet, der Funktionsaufruf erfolgt allerdings ohne Angabe derselben. Ferner können Templates Standardwerte für Parameter definieren, so dass die Übergabe optional ist und in diesem Fall der vordefinierte Wert genommen wird. Funktionsaufrufe müssen jedoch alle Parameter enthalten. Wie Template-Regeln haben auch diese Top-Level-Deklarationen einen Sequenzkonstruktor, der die Rückgabe definiert.

2.7.6 Variablen und temporäre Bäume

Für XSLT sind alle Variablen Konstanten. In Java werden diese mit dem Schlüsselwort `final` deklariert. Variablen können in Stylesheets eine Sequenz referenzieren und somit alle Werte repräsentieren, die das Datenmodell von XPath definiert.

Eine Variablendeklaration kann über einen Pfadausdruck oder mit Hilfe eines Sequenzkonstruktors mit einem Wert initialisiert werden. Letzterer kann Ergebniselemente und Instruktionen enthalten. Das `as`-Attribut der Deklaration legt den Sequenztyp mit Hilfe eines Typdeskriptors fest. Ist es nicht gesetzt, wird ein temporärer Baum erzeugt. Der Baum kann als Eingabe für weitere Template-Regeln dienen. Auf diese Weise lassen sich Stylesheets mit mehreren Phasen realisieren. Beispielsweise kann eine Transformation das Ergebnis eines ersten

Durchlaufs, durch ein Eingabedokument, in einen temporären Baum umleiten. Dieser kann als Einabe für die zweite Phase dienen.

2.7.7 Stylesheet-Module

Transformationen sind leicht erweiterbar. Ein Stylesheet kann durch `<xsl:import>`- und `<xsl:include>`-Deklarationen weitere XSLT-Dokumente als *Stylesheet-Module* einbinden. Bei einem Import ist die Priorität einer importierten Regel niedriger als die Regeln des importierenden Stylesheets. Auf diese Weise kann ein Stylesheet importierte Regeln überschreiben. Im Gegensatz dazu interpretiert ein XSLT-Prozessor ein, durch `<xsl:include>` eingebundenes, XSLT-Dokument als wäre der Inhalt in das einbindende Stylesheet kopiert worden.

2.7.8 Ausführung einer Transformation

Dieser Abschnitt zeigt anhand des Beispiels in Listing 2.11, wie eine Transformation ausgeführt wird. Das Stylesheet soll nicht als Vorbild, für gute Transformations-Programmierung dienen. Vielmehr soll es am Beispiel eines vollständigen XSLT-Skripts das Zusammenspiel der Komponenten veranschaulichen. XSLT ist für Programmierer aus dem Umfeld der imperativen Sprachen nicht intuitiv. Der deklarative Charakter der Sprache soll hier näher gebracht werden, in dem die Wirkung eines einfachen Skripts genauer beschrieben wird.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsl:stylesheet
3    xmlns:xsl=
4    "http://www.w3.org/1999/XSL/Transform"
5    version="2.0">
6
7   <xsl:template match="/">
8    <xsl:variable name="title"
9    select="text/title[1]"/>
10   <html>
11    <head>
12     <title>
13      <xsl:value-of select="$title"/>
14     </title>
15    </head>
16    <body>
17     <h1><xsl:value-of select="$title"/></h1>
18     <xsl:apply-templates/>
```

```
19      </body>
20     </html>
21    </xsl:template>
22
23    <xsl:template match="title"/>
24
25    <xsl:template match="paragraph">
26     <p>
27      <xsl:apply-templates/>
28     </p>
29    </xsl:template>
30
31    <xsl:template match="bold">
32     <b>
33      <xsl:apply-templates/>
34     </b>
35    </xsl:template>
36
37    <xsl:template match="italic">
38     <i>
39      <xsl:apply-templates/>
40     </i>
41    </xsl:template>
42
43   </xsl:stylesheet>
```

Listing 2.11: Beispiel-Stylesheet

Die Transformation folgt, bis auf die Verarbeitung des `<title>`-Elements, dem regelbasierten Entwurfsmuster. Es transformiert ein relativ unstrukturiertes XML-Dokument mit einigen Strukturinformationen in ein HTML-Dokument. In Listing 2.12 ist ein Beispieldokument vor der Transformation dargestellt.

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <text>
3  <title>Hello, World!</title>
4  <paragraph>
5   Dies ist ein <bold>Paragraph</bold>, der
6   in <italic>HTML</italic> übersetzt werden
7   soll.
8  </paragraph>
9 </text>
```

Listing 2.12: Eingabedokument

Die Template-Regel mit dem Muster «/» verarbeitet den Dokumentknoten des Quelldokuments. Sie beginnt mit der Deklaration der Variable `$title`. Die Variable referenziert das erste Kind des Dokumentelements mit dem Namen `<title>`. Ergebniselemente bilden den HTML-Rahmen. Dem `<title>`-Element im Kopf des Dokuments wird der Wert der `$title`-Variable zugewiesen. Im Inhalt-Teil[2] wird der Titel ferner in eine große Überschrift[3] eingebettet. Bis zu diesem Punkt entspricht das Stylesheet dem navigational Design-Pattern. Mit der `<xsl:apply-templates>`-Instruktion beginnt der regelbasierte Teil. Sie veranlasst, dass für jeden Kindknoten in Dokumentreihenfolge eine Template-Regel gewählt und ausgeführt wird. Im Falle von Textknoten wird eine integrierte Regel verwendet, die den Knoten in das Ausgabedokument kopiert.

Die `<title>`-Elemente des Quelldokuments werden bereits in der Regel für den Dokumentknoten bearbeitet. Die entsprechende Template-Regel erzeugt daher keine Ausgabe. In diesem Beispiel wird davon ausgegangen, dass es nur ein Titelelement gibt, da alle weiteren ignoriert würden. Die übrigen Template-Regeln übersetzen die `paragraph`-, `bold`- und `italic`-Elemente in die HTML-Elemente für Paragraphen, Fett- und Kursivschrift. Das transformierte HTML-Dokument für das Beispieldokument zeigt Listing 2.13.

```
1  <html>
2   <head>
3    <meta http-equiv="Content-Type"
4      content="text/html;␣charset=ISO-8859-1">
5     <title>Hello, World!</title>
6   </head>
7   <body>
8    <h1>Hello, World!</h1>
9    <p>
10   Dies ist ein <b>Paragraph</b>,
11   der in <i>HTML</i> übersetzt
12   werden soll.
13   </p>
14  </body>
15 </html>
```

Listing 2.13: Ausgabedokument

[2] Inhalt des `<body>`-Elements
[3] Entspricht dem `<h1>`-Tag in HTML

3 Lightweight-Declarative-Coordination

Im Rahmen dieser Diplomarbeit wurde das Tree-Transformer-Plugin für jABC entwickelt werden, das eine graphische Beschreibung von Baumtransformationen ermöglicht. Dieses Kapitel geht näher auf das Konzept ein.

Tree-Transformer dient als Ersatz für einen XSLT-Editor und bietet einen vereinfachten Entwurfsprozess. Die Modellierungsumgebung ist so restriktiv, dass viele Fehlerquellen bei der Erstellung von XSLT-Stylesheets erkannt werden. Es ist vorgesehen, dass der Erstellungsprozess einer Transformation auf mehrere Entwickler verteilt werden kann. Die Verteilung der Aufgaben soll die Effizienz erhöhen und eine Trennung nach Kompetenzen ermöglichen.

Die Hauptanforderungen an das Plugin sind:

- Transformationsgraphen sollen einfach in ausführbare SIB-Graphen eingebunden werden können.
- Die Transformationsgraphen sollen eine Transformation einfach und abstrakt darstellen.
- Es ist eine möglichst große Teilmenge des Funktionsumfangs von XSLT abzubilden.

In dem Entwurfsprozess von ausführbaren SIB-Graphen mit jABC werden typischerweise zwei Nutzergruppen unterschieden.

Anwendungsexperte:
Der Anwendungsexperte hat fachliche Kenntnis auf dem Gebiet der Zielanwendung. Er modelliert ausführbare SIB-Graphen und benötigt kein technisches Verständnis.

SIB-Experte:
Die Komponenten eines ausführbaren SIB-Graphen werden von einem SIB-Experten implementiert. Er hat Detailkenntnis in der Zielsprache.

Bei der Erstellung von Transformationsgraphen ergeben sich aus den Anforderungen drei Rollen, in die ein Entwickler eingeordnet werden kann.

Anwendungsexperte:
Die Transformationsgraphen werden in ausführbaren Graphen aufgerufen. Der Anwendungsexperte des LPC-Konzepts bildet somit die Schnittstelle zwischen dem LPC-Konzept und dem Konzept des Tree-Transformer-Plugins.

Transformationsdesigner:
Ein Transformationsdesigner hat ein Verständnis von Transformationen, soll jedoch keine Detailkenntnisse in XSLT haben müssen. Er ist für die Komposition der Transformationskomponenten zuständig.

XSLT-Experte:
Der XSLT-Experte kennt sich im Detail mit der Sprache XSLT aus. Er benötigt im Idealfall keinen Überblick über die Transformation, sondern implementiert ihre Komponenten oder Module.

3.1 Auswahl des Konzepts

Die Anforderungen an Tree-Transformer haben zu unterschiedlichen Überlegungen geführt, wie diese erfüllt werden können. Die folgenden Unterkapitel beschreiben drei mögliche Konzepte und ihre Vor- und Nachteile. Anschließend wird die Wahl des verwendeten Konzepts begründet.

3.1.1 Koordination in ausführbaren SIB-Graphen

Der erste Lösungsansatz setzt die Koordination von Stylesheet-Modulen in ausführbare SIB-Graphen um. Bei diesem Ansatz werden Stylesheet-Module von einem XSLT-Experten in einem XSLT-Editor erstellt. Ein ausführbarer SIB kapselt den Aufruf der Verarbeitungs-API, wie zum Beispiel JAXP, um das Stylesheet-Modul mit einem XSLT-Prozessor auszuführen. Das Modul und das Eingabedokument werden aus dem gemeinsamen Speicher gelesen und die Ausgabe wird in den Speicher gelegt. Die Dokumente können über ausführbare *Common-SIBs* vom Dateisystem gelesen sowie in eine Datei geschrieben werden. Besteht eine Transformation aus mehreren Modulen, werden diese jeweils in einem ausführbaren SIB-Graphen über den oben genannten SIB aufgerufen. Mit weiteren ausführbaren SIBs können Verzweigungen und Schleifen modelliert werden, die eine Koordination der Transformation auf Modellebene erlauben.

Abbildung 3.1 zeigt, wie ein ausführbarer SIB-Graph aussehen könnte, der die Koordination von Stylesheet-Modulen realisiert. Das Start-SIB, gekennzeichnet durch einen unterstrichenen Bezeichner, liest das Eingabedokument ein und legt es unter dem Schlüssel `$input` in den gemeinsamen Speicher. Das manuell erstellte Stylesheet wird auf die gleiche Weise geladen. Das Stylesheet wird anschließend aufgerufen. Das Ausgabedokument wird im gemeinsamen Speicher

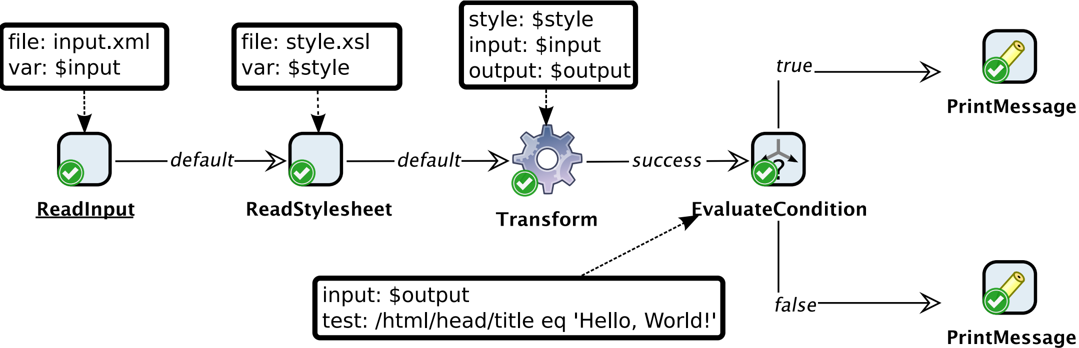

Abbildung 3.1: Modellierung einer Transformation in LPC-Graphen

gehalten. Der SIB mit dem Bezeichner «`EvaluateCondition`» liest das Ausgabedokument ein und führt einen XPath-Ausdruck darauf aus, der auf dem XML-Dokument zu dem boolschen Wert `true` oder `false` evaluiert wird. Über den entsprechenden Branch mit der Bezeichnung «`true`» oder «`false`» wird das nachfolgende SIB gewählt. In diesem Beispiel wird jeweils eine Ausgabe in die Konsole geschrieben. Die Verzweigung könnte in einem umfassenderen Beispiel auch auf weitere Transformationen zeigen.

Dieser Ansatz kann einfach umgesetzt werden, da keine neue Graphsyntax für die Modellierung nötig ist. Weiterhin muss ein Transformationsdesigner keinen neuen Modellierungsansatz erlernen.

Durch die Verlagerung der Koordinierung von Stylesheet-Modulen in das LPC-Konzept müssen der Transformationsdesigner und der Anwendungsexperte gemeinsam den zugehörigen SIB-Graphen bearbeiten. Das ist notwendig, um einerseits die Komponenten der Transformation zu koordinieren (Transformationsdesigner) und andererseits die Anwendung zu modellieren (Anwendungsexperte). Dieser Nachteil lässt sich durch Hierarchisierung minimieren. Das bedeutet, dass der Transformationsdesigner die Beschreibung einer Transformation in einen Untergraphen kapselt, die der Anwendungsexperte in seinen Kontrollflussgraphen mit einem Graph-SIB einbindet.

Der Ansatz mischt das prozedurale LPC-Konzept mit dem deklarativen Ansatz von XSLT. Dadurch werden die Funktionen zur Einbindung von Stylesheet-Modulen und die Realisierung von Transformationsphasen von XSLT zu einer Verarbeitungs-API verlagert. Jeder Aufruf einer Stylesheet-Komponente erzeugt ein Zwischenergebnis, das im gemeinsamen Speicher abgelegt wird. So kann die Ausgabe einer Transformation als Eingabe für die nächste dienen. Ferner benötigen die Kontrollstrukturen in den ausführbaren Graphen Zugriff auf die Baumstruktur der XML-Dokumente, um zum Beispiel eine Verzweigung aufgrund von Eigenschaften des Dokumentes zu realisieren. Dies erfordert, dass das entsprechende XML-Dokument über eine API, wie zum Beispiel dem DOM,

zugänglich gemacht werden muss. Ein XSLT-Prozessor muss jedes Stylesheet-Modul getrennt aufrufen, da sie in einem ausführbaren SIB gekapselt und nicht in ein zusammengesetztes Stylesheet übersetzt werden.

Ferner ist das Plugin bei diesem Lösungsansatz kein Ersatz für einen XSLT-Editor. Die Komponenten müssen weiterhin in einem XSLT-Editor erstellt werden.

3.1.2 Unterteilung in zwei Modellierungsebenen

Das zweite Konzept sieht die Entwicklung einer Graphsyntax, die deklarativ ist und von XSLT abstrahiert, vor. Hierbei muss ein Mittelweg gefunden werden zwischen der Mächtigkeit der Transformationsgraphen und dem Grad der Abstraktion. Es ist jedoch schwierig, einen derartigen Kompromiss zu finden, da eine höhere Abdeckung der Funktionen von XSLT auch mit einer geringeren Abstraktion einhergeht. Somit muss ein Modellierer detailliertere Kenntnisse in der Zielsprache haben. Abstrahieren die Graphen stark von XSLT, kann dies durch den Verzicht auf Kontrollstrukturen realisiert werden. Auf diese Weise werden die Möglichkeiten der Transformationsgraphen eingeschränkt. Weiterhin kann eine abstrakte Darstellung durch eine aufwändigere Transformation der Graphen in die Zielsprache erreicht werden. Der Pluginentwickler definiert Funktionen in XSLT vor, die der Modellierer in den Transformationsgraphen über ein SIB einbinden kann. Ein großer Funktionsumfang benötigt in diesem Fall viele vordefinierte SIBs, die jeweils eine Aufgabe übernehmen.

In Abbildung 3.2 ist ein triviales Beispiel zu sehen, wie ein Transformationsgraph aussehen könnte, der vordefinierte Funktionen verwendet, um von XSLT zu abstrahieren. Der Graph hat die gleiche Semantik, wie das XSLT-Stylesheet

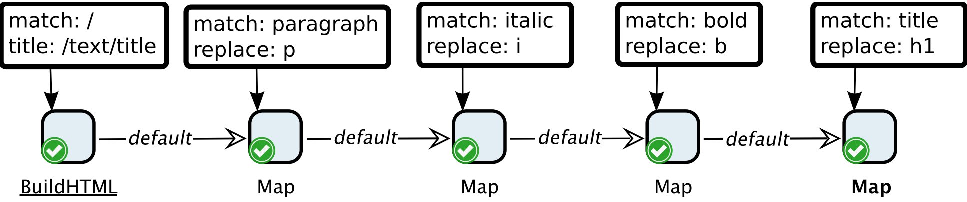

Abbildung 3.2: Modellierung einer Transformation in einem Transformationsgraphen mit vordefinierten Funktions-SIBs für einzelne Aufgaben

aus Listing 2.11 auf Seite 38. Er beinhaltet zwei verschiedene Typen von SIBs. Beide verarbeiten die Knoten, die auf das Muster des SIB-Parameters `match` passen. Der `BuildHTML`-SIB ist für den Dokumentknoten des Eingabedokuments definiert und erzeugt ein HTML-Gerüst. Der SIB-Parameter `title` wählt den Knoten des Eingabedokuments, dessen Wert in der Titelleiste des Browsers angezeigt wird. Die nachfolgenden `Map`-SIBs verarbeiten jeweils ein XML-Element

und ersetzen ihn durch eines mit der Bezeichnung, die in dem SIB-Parameter `replace` angegeben wird. In diesem Beispiel werden `<paragraph>`-, `<bold>`-, `<italic>`- oder `<title>`-Elemente in die HTML-Tags `<p>`, `<b>`, `<i>` oder `<h1>` übersetzt.

Dieser Lösungsansatz bietet den Vorteil, dass die Transformationen nicht in Kontrollflussgraphen modelliert werden. Weiterhin kann ein Transformationsdesigner Tree-Transformer als Ersatz für einen XSLT-Editor verwenden. Es wird darauf verzichtet, Stylesheet-Module in Graphen einzubinden. Auf diese Weise steht nicht die volle Funktionalität von XSLT zur Verfügung. Es wird jedoch kein XSLT-Experte benötigt.

Die verwendeten SIBs sind auf diesen Anwendungsfall spezialisiert. Der `BuildHTML`-SIB muss für weitere Ausgabeformate, wie zum Beispiel XSL-FO, durch einen anderen SIB ersetzt werden. Für komplexere Aufgaben als die Ersetzung von XML-Elementen durch andere XML-Elemente werden, zusätzlich zu dem `Map`-SIB, weitere SIBs benötigt. Es wird eine Abstraktion von XSLT erreicht, um den Preis, dass die Komplexität durch eine große SIB-Palette steigt.

3.1.3 Unterteilung in drei Modellierungsebenen

Bei dem dritten Konzept wird der Entwurfsprozess in drei Modellierungsebenen unterteilt, die jeweils einer der genannten Rollen zugeordnet ist.

Ausführbare Graphen:
Der Anwendungsexperte bindet Transformationsgraphen in Kontrollflussgraphen über das Konzept der Hierarchisierung ein.

TT-Graphen:
Eine abstrakte Darstellung einer Transformation in *Tree-Transformer-Graphen* (TT-Graphen) dient dem Transformationsdesigner zur Koordinierung von Transformationskomponenten.

XSL-Graphen:
Die Komponenten eines Transformationsgraphen werden von einem XSLT-Experten in XSL-Graphen erzeugt. Sie stellen eine kanonische Abbildung von XSLT auf Graphen dar.

Die Unterteilung der Transformationsgraphen in zwei Typen resultiert aus den Eigenschaften der Zielsprache XSLT. Einerseits lässt sich XSLT aufgrund ihrer Notation in XML einfach als Graph darstellen, andererseits soll die graphische Modellierung den Vorteil der Abstraktion und Vereinfachung haben. Die Unterteilung in eine kanonische Abbildung der Zielsprache (unterste Ebene) und eine

abstrakte Darstellung von Transformationen mit einer eigenen Semantik (zweite Ebene) ermöglicht, von beiden Ansätzen zu profitieren. Beide Graph-Typen müssen jedoch in XSLT übersetzt werden.

Diese Lösung erfüllt als einzige der drei angeführten Konzepte die drei Hauptanforderungen:

1. Transformationsgraphen werden über einen speziellen Graph-SIB in LPC-Graphen eingebunden. Das Konzept der Hierarchisierung ist einem LPC-Nutzer bekannt.

2. Die TT-Graphen abstrahieren von XSLT und binden XSL-Graphen als Komponenten ein. So steht TT-Graphen die volle Funktionalität von XSLT zur Verfügung.

3. Die XSL-Graphen können XSLT vollständig abbilden.

Weiterhin ist der Ansatz für LPC-Nutzer und XSLT-Experten intuitiv und es wird eine Aufgabenteilung durch die Trennung der Modellierungsebenen erreicht.

Da das dritte Konzept als einziges alle Funktionen von XSLT zur Verfügung stellt und gleichzeitig eine abstrakte Graphdarstellung für die Koordination von Transformationskomponenten anbietet, wird es für das Tree-Transformer-Plugin verwendet.

Das Konzept heißt *Lightweight-Declarative-Coordination* (LDC). Dabei handelt es sich um eine Erweiterung des LPC-Konzepts. LDC wird im Folgenden näher beschrieben.

In Abbildung 3.3 sind die drei Modellierungsebenen zu sehen. Die oberste Ebene (LPC-Ebene) ermöglicht die Ausführung von Transformationen. Sie verwendet die bestehenden Funktionalitäten von jABC. Eine Transformation wird über einen `Transform`-SIB aufgerufen. Der gemeinsame Speicher verwaltet das Eingabe- und Ausgabedokument. Ein Modellierer auf dieser Ebene ist ein Anwendungsexperte. Die Einbindung einer Baumtransformation wird durch den `Transform`-SIB als eine Komponente gekapselt, die für den Anwender der LPC-Ebene transparent ist.

Die beiden unteren Ebenen sind in Abbildung 3.3 von der obersten getrennt. Dies spiegelt wider, dass die oberste Ebene aus Kontrollflussgraphen besteht, die auf generische, imperative Sprachen, wie zum Beispiel Java, abgebildet werden, während die beiden unteren Ebenen die deklarative Sprache XSLT kapseln.

Die zweite Ebene bilden die Tree-Transformer-Graphen (TT-Graphen). Sie stellen eine Abstraktion von XSLT dar. Sie kapseln Komponenten der XSL-Ebene und TT-Graphen in spezielle Graph-SIBs. Ein TT-Graph repräsentiert entweder eine XSL-Funktion oder eine Template-Regel. Die Graphen der zweiten Ebene werden in XSLT übersetzt und mit einem XSLT-Prozessor ausgeführt.

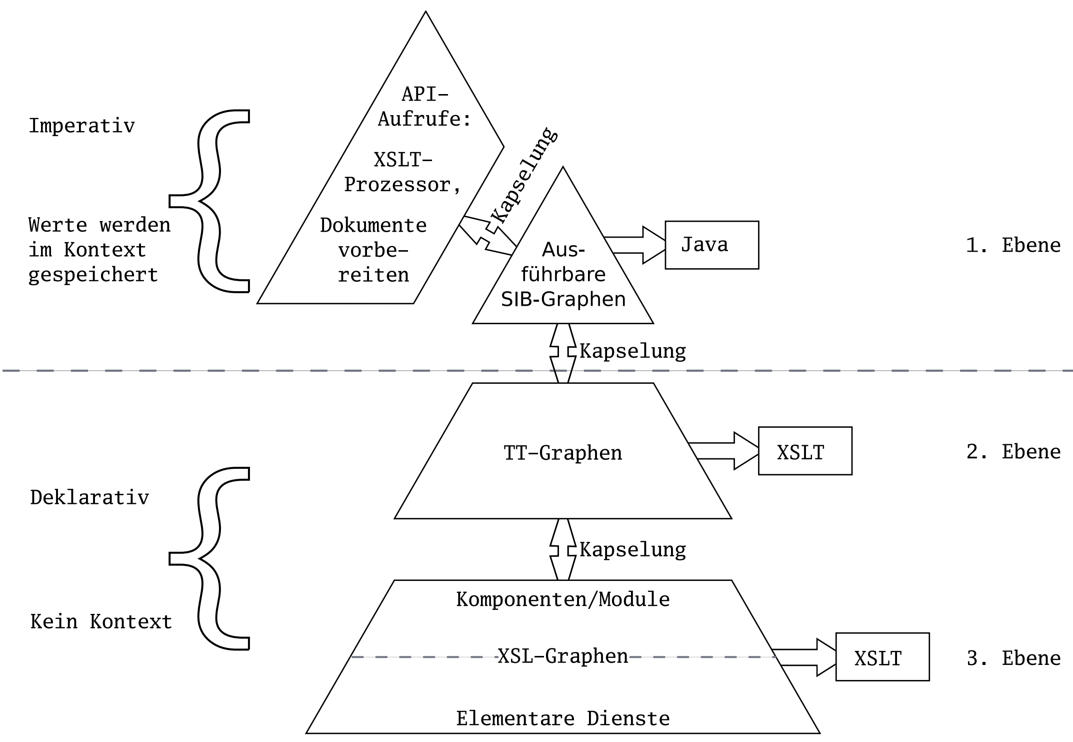

Abbildung 3.3: Ebenen der Modellierung

Die Graphen der LPC-Ebene binden den Haupt-TT-Graphen einer Transformation an einen `Transform`-SIB. So kann eine Transformation in Kontrollflussgraphen ausgeführt werden. Der Aufruf eines XSLT-Prozessors wird in dem `Transform`-SIB gekapselt. Ein Modellierer der zweiten Ebene ist ein Transformationsdesigner.

Die unterste Ebene bilden die XML-Graphen. Sie stellen eine direkte Abbildung von XML auf Graphen dar. Da XSLT in XML notiert wird, lässt sich ein Stylesheet auf diese Weise graphisch darstellen. Repräsentiert ein XML-Graph eine Transformation, wird er XSL-Graph genannt. Bei der Erstellung von Graphen der dritten Ebene wird nicht von der Zielsprache abstrahiert. Die Bezeichnung „graphische Programmierung“ wäre somit treffender, als „Modellierung“. Es wird jedoch einheitlich der Begriff Modellierung für alle drei Ebenen verwendet.

Ausführbare Graphen koordinieren Komponenten zu einer Anwendung. Dabei werden die Komponenten nicht modelliert, sondern durch einen SIB-Experten in einer Zielsprache implementiert. Das hat unter anderem folgende Gründe:

- Bei der Generierung einer Zielanwendung ist die Zielsprache austauschbar. Ein ausführbarer SIB-Graph kann zum Beispiel einerseits in eine Anwendung für *Java-Micro-Edition* (JME) und andererseits in eine Anwendung

für Java generiert werden. Die Applikation kann so auf einem eingebetteten System und jeder Plattform, die Java unterstützt, ausgeführt werden. Die graphische Programmierung jedes einzelnen Statements würde erfordern, einen SIB-Graphen für jede Zielsprache zu erstellen.

- Ausführbare SIB-Graphen werden in generische, imperative Sprachen, wie zum Beispiel Java oder C#, übersetzt. In diesen Sprachen werden viele Funktionen in Bibliotheken angeboten.
- LPC-Graphen werden von einem Anwendungsexperten modelliert, der kein technisches Verständnis benötigt.

Im Gegensatz zu den grob granularen Graphen des LPC-Konzepts werden bei dem LDC-Konzept die Graphen bis hin zu der einzelnen XSLT-Instruktion ausmodelliert. XSL-Graphen stellen somit sowohl elementare Dienste als auch Komponenten zur Verfügung. Die folgenden Argumente rechtfertigen diese Entscheidung:

- Stylesheets werden als XML-Dokumente notiert. Das ermöglicht eine einfache Darstellung als Baum.
- Die Zielsprache von Transformationsgraphen ist auf XSLT festgelegt.
- Ein XSLT-Experte kann Stylesheets manuell erstellen und über die Importfunktion von Tree-Transformer in XSL-Graphen konvertieren. Die XSL-Graphen sind lediglich eine Zusatzfunktion.
- XSLT ist auf Transformationen spezialisiert und wird über einen Interpreter (XSLT-Prozessor) ausgeführt. Es müssen weder Bibliotheken importiert werden noch werden Stylesheets kompiliert.
- Tree-Transformer soll ein Ersatz für einen XSLT-Editor sein.
- Tree-Transformer kann für den gesamten Entwurfsprozess verwendet werden.

In der darüberliegenden Ebene werden die XSL-Graphen als Komponenten verwendet, die entweder eine Template-Regel oder eine XSL-Funktion repräsentieren. Ein Modellierer der untersten Ebene ist ein XSLT-Experte. Er entspricht dem SIB-Experten des LPC-Konzepts, da er die Komponenten zur Verfügung stellt, die in SIBs der zweiten Ebene gekapselt werden. Er implementiert jedoch keine neuen SIB-Klassen, da er die Komponenten einer Transformation durch Graphen beschreibt.

XSLT vollständig in Graphen zu übersetzen, bedeutet nicht die Abkehr von der Serviceorientierung. Auf der ersten und zweiten Ebene werden weiterhin Services koordiniert. Lediglich die XSL-Ebene bildet eine Ausnahme.

In den Kapiteln 3.2, 3.3 und 3.4 werden die einzelnen Ebenen der Modellierung beleuchtet.

3.2 Ausführung von Transformationen

Transformationen werden über LPC-Graphen ausgeführt. Tree-Transformer ermöglicht eine einfache Einbindung von Transformationsgraphen. Dies ist für einen Anwendungsexperten intuitiv und setzt keine XSLT-Kenntnisse oder technisches Verständnis über Baumtransformationen voraus.

In dem ausführbaren Graph in Abbildung 3.4 wird die Einbindung eines Transformationsgraphen anhand der Beispieltransformation aus Kapitel 2.7.8 auf Seite 38 gezeigt. Die Ausführung beginnt in dem `ReadTextFile`-SIB mit

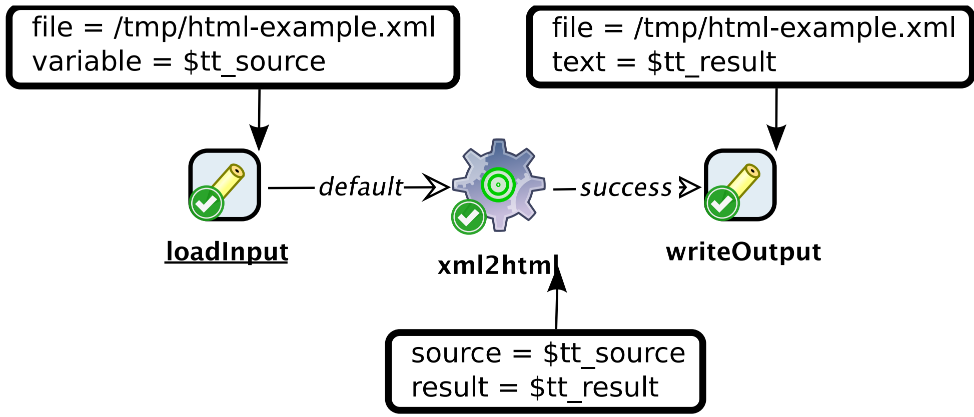

Abbildung 3.4: Beispiel eines ausführbaren Graphen

dem Label «`loadInput`». Er gehört zu den Common-SIBs. Der SIB-Parameter `file` legt die Quelldatei auf «`/tmp/html-example.xml`»[1] fest. Der Text der Datei wird unter `$tt_source` im gemeinsamen Speicher abgelegt. Der SIB mit der Bezeichnung «`xml2html`» ist ein `Transform`-SIB, der einen TT-Graphen referenziert. Der TT-Graph repräsentiert die Transformation aus Kapitel 2.7.8. Das Eingabedokument liest er aus der Variablen `$tt_source` des gemeinsamen Speichers. Die Ausgabe schreibt er in `$tt_result`. Anschließend serialisiert der `WriteTextFile`-SIB mit der Bezeichnung «`writeOutput`» die Ausgabe in die Datei «`/tmp/html-example.html`». Alle drei SIB-Instanzen exportieren den `error`-Branch. Weiterhin veröffentlicht der SIB mit der Bezeichnung «`writeOutput`» den `default`-Branch. Der Name der entsprechenden SIB-Instanzen ist fett geschrieben. So wird entweder durch einen Fehler oder die erfolgreiche Ausführung des SIBs «`writeOutput`» der aktuelle Graph beendet.

[1]In einem richtigen Projekt sollte ein plattformunabhängiger und relativer Pfad gewählt werden.

Ein Anwendungsexperte benötigt nur den `Transfom`-SIB, um Transformationsgraphen auszuführen. Dieser SIB ist ein spezieller Graph-SIB, der einen XSL- oder TT-Graph referenziert. Das Eingabedokument liest er aus dem gemeinsamen Speicher. Das Dokument kann auf verschiedene Weise eingelesen werden. Im einfachen Fall liegt es, wie in dem Beispiel, als Textdatei im lokalen Dateisystem. Hierfür kann der `ReadTextFile`-SIB verwendet werden.

Das Ausgabedokument legt der `Transform`-SIB ebenfalls in den gemeinsamen Speicher. Es kann mit dem Common-SIB `WriteTextFile` in eine Datei serialisiert werden.

Der `Transform`-SIB wird über das SIB-Adapter-Entwurfsmuster realisiert. Er kapselt Aufrufe der JAXP-API, über die der XSLT-Prozessor angesprochen wird. In einem ersten Schritt wird der referenzierte Transformationsgraph in ein XSLT-Stylesheet übersetzt. Anschließend verarbeitet das Stylesheet das Eingabedokument. Die Implementierung wird in Abschnitt 4.5.4 genauer beschrieben.

Eine Transformation ist auf der LPC-Ebene eine Komponente, die durch den `Transform`-SIB gekapselt wird. Es werden nur die vorhandenen Funktionen, die einem LPC-Nutzer bekannt sind, verwendet. So können Baumtransformationen nahtlos in die Kontrollflussgraphen von jABC integriert werden.

3.3 Tree-Transformer-Graph

TT-Graphen beschreiben abstrakt eine Baumtransformation. Sie werden von einem Transformationsdesigner modelliert. Die Erstellung eines TT-Graphen soll keine Detailkenntnisse in XSLT erfordern. Komponenten werden über weitere Transformationsgraphen (TT- oder XSL-Graphen) realisiert und sollen einfach eingebunden werden können. Die Graphen müssen übersichtlich und klar strukturiert sein. Die Komponenten sollen sich untereinander möglichst wenig beeinflussen können. Innerhalb einer Komponente ist eine starke Abhängigkeit der Hierarchieebenen hingegen erwünscht. Diese Differenzierung von loser Kopplung von Komponenten und starker Kopplung innerhalb einer Komponente ist ein Grundprinzip der Serviceorientierung. Auf diese Weise ist die Wirkung eines TT-Graphen lokal und es ist einfacher für den Modellierer, den Überblick über eine umfangreiche Transformation zu behalten.

Das Modellierungskonzept der TT-Graphen basiert darauf, dass alle Werte im XPath-2.0-Datenmodell eine Sequenz sind und alle Instruktionen und zusammengesetzten Konstrukte der Sprache auf einer unveränderlichen Eingabe operieren und eine Sequenz als Rückgabe haben. Nachfolgend wird das Konzept dieser Modellierungsebene näher beschrieben. In Abschnitt 3.3.1 werden die Verifikationsmethoden für Graphen der TT-Ebene dargestellt. Abschnitt 3.3.2 erklärt an einem Beispiel den Aufbau von TT-Graphen. In den Abschnitten 3.3.3, 3.3.4 und 3.3.5 werden die Komponenten der TT-Graphen näher beschrieben.

Abschließend fasst Abschnitt 3.3.6 die Ergebnisse zusammen.

3.3.1 Verifikationsmethoden

Der LocalChecker prüft während der Modellierung lokale Bedingungen an den SIB-Instanzen. So werden Fehlerquellen frühzeitig entdeckt und können behoben werden. Bedingungen, die der LocalChecker an TT-Graphen prüft, sind unter anderem:

- Eine `result`- oder unbeschriftete Kante darf nur auf ein `XPath`-, `WithParameter`-, `If`-, `Group`- oder `Output`-SIB zeigen. Alle anderen SIBs können keine Eingabe verarbeiten.
- Die `true`- und `false`-Achsen eines `If`-SIBs zeigen immer auf einen weiteren `If`-SIB oder einen `DocumentNode`-SIB.
- Der SIB-Parameter `match` eines `TemplateMain`- oder `FunctionMain`-SIBs und der SIB-Parameter `test` eines `If`-SIBs müssen gesetzt sein.

GEAR kann globale Eigenschaften eines Modells sichern. Dabei steht nicht die syntaktische Korrektheit eines Graphen im Vordergrund, sondern semantische Eigenschaften eines speziellen Graphen.

3.3.2 Beispiel eines TT-Graphen

Das XSLT-Stylesheet in Listing 3.1 ist ein Teil des Beispiels aus Kapitel 2.7.8 auf Seite 38. Die Erzeugung des HTML-Rahmens und die Verarbeitung des `<title>`-Elements wurden hier aus Platzgründen weggelassen.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsl:stylesheet
3    xmlns:xsl=
4    "http://www.w3.org/1999/XSL/Transform"
5    version="2.0">
6    <xsl:template match="paragraph">
7      <p>
8        <xsl:apply-templates/>
9      </p>
10   </xsl:template>
11   <xsl:template match="italic">
12     <i>
13       <xsl:apply-templates/>
14     </i>
15   </xsl:template>
```

```
16    <xsl:template match="bold">
17      <b>
18        <xsl:apply-templates/>
19      </b>
20    </xsl:template>
21  </xsl:stylesheet>
```

Listing 3.1: Vereinfachte Variante des Stylesheets aus Listing 2.11 auf Seite 38

Das Stylesheet transformiert das Eingabedokument aus Listing 2.12 auf Seite 39, in folgende Ausgabe:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2  Hello, World!
3
4   Dies ist ein <b>Paragraph</b>, der
5   in <i>HTML</i> übersetzt werden soll.
```

Listing 3.2: Ausgabedokument

Das Stylesheet soll als einfaches Beispiel dienen, das im Folgenden als TT-Graph realisiert wird. Es erzeugt weder ein wohlgeformtes XML-Dokument noch ein HTML-Dokument. In Abbildung 3.5 ist ein TT-Graph zu sehen, der die gleiche Semantik hat, wie das Stylesheet aus Listing 3.1. In ihm werden die Template-

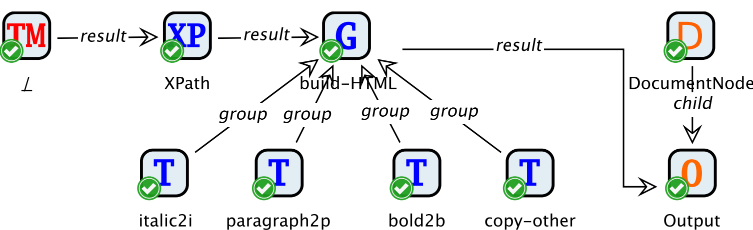

Abbildung 3.5: Beispiel eines TT-Graphen

Regeln für die Übersetzung der `paragraph`-, `bold`-, und `italic`-Elemente in die zugehörigen HTML-Elemente in Transformationskomponenten realisiert. Dieser Graph wird als Test für das Plugin verwendet. Für dieses Beispiel wird hingegen der Graph in Abbildung 3.6 eingesetzt, der die gleiche Semantik hat. Bei ihm wird auf die Einbindung von weiteren Transformationskomponenten und die Hierarchisierung durch Graph-SIBs verzichtet, damit das Beispiel in sich geschlossen ist. Weiterhin kann auf diese Weise der Aufbau von TT-Graphen anhand eines einfachen Beispiels erklärt werden.

Die farbigen Rahmen teilen den Graphen in vier Bereiche, die voneinander getrennt sind. Das rote Rechteck umfasst die Deklaration der Eingabe. Die blaue

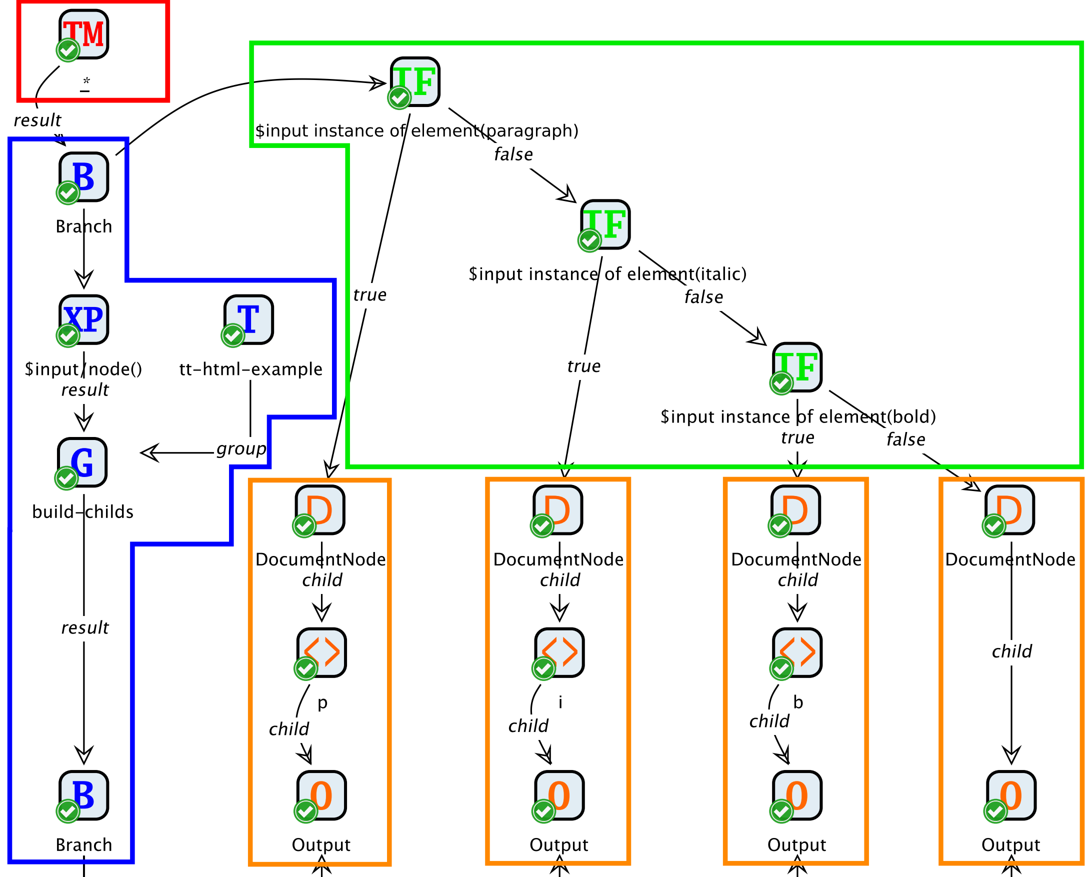

Abbildung 3.6: Beispiel eines TT-Graphen

Markierung rahmt die Verarbeitung der Eingabe ein. Der in grün eingefasste Bereich entscheidet, welche Ausgabe zurückgegeben werden soll. Schließlich dienen die orange umrandeten Teile des Graphen der Erzeugung der möglichen Ausgaben. Jeder TT-SIB lässt sich in einen dieser Bereiche einordnen. Das SIB-Icon der SIBs ist entsprechend eingefärbt, um Vermischungen der Bereiche vorzubeugen. Nachfolgend wird der Aufbau eines TT-Graphen an diesem Beipiel genauer erläutert.

Deklaration der Eingabe (rot):

Der `TemplateMain`-SIB mit der Bezeichnung «*» ist der Haupt-SIB des Graphen. Er wird immer als Start-SIB markiert. Das visualisiert jABC, indem der Bezeichner eines Start-SIBs unterstrichen wird. Der `TemplateMain`-SIB legt fest, dass dieser TT-Graph eine Template-Regel definiert. Das Label entspricht dem Pattern, das die Knoten definiert, für die diese Regel angewendet wird. In dem Beispiel bedeutet der Bezeichner «*», dass die Template-Regel alle Elementknoten des Dokuments verarbeitet.

Verarbeitung der Eingabe (blau):
Wird die Regel aufgerufen, steht der Kontextknoten, in diesem Fall immer ein Elementknoten, als Eingabe zur Verfügung. Diese wird über die `result`-Achse des SIBs an einen `Branch`-SIB gegeben. Der `Branch`-SIB verteilt die Eingabe einerseits an den obersten `If`-SIB und andererseits an einen `XPath`-SIB. Letzterer ist Teil der Verarbeitung der Eingabe (blauer Rahmen). Ein `XPath`-SIB wertet einen Pfadausdruck aus. Die Eingabesequenz wird aus den eingehenden Kanten gebildet und von der Variablen `$input` referenziert. Der Bezeichner entspricht dem Pfadausdruck des SIBs: «`$input/node()`». Dieser wählt alle Kindknoten der Eingabesequenz. Diese entspricht dem Kontextknoten.

Die Kindknoten des Elements werden über die `result`-Kante an einen `Group`-SIB weitergereicht. Dieser sucht für jeden der Kindknoten eine Template-Regel, die den Knoten verarbeiten kann und ruft sie auf. In diesem Fall steht nur eine Template-Regel zur Verfügung, die über die `group`-Achse an den `Group`-SIB gebunden wird. Die Template-Regel referenziert den aktuellen Transformationsgraphen. Der rekursive Aufruf verarbeitet somit alle Elementknoten, die Kinder des Kontextknotens sind. Wird für einen Knoten keine Regel gefunden, werden die in der Zielsprache XSLT definierten, integrierten Template-Regeln angewendet. Eine dieser Regeln kopiert zum Beispiel alle Textknoten. Die Ausgabesequenz der Gruppe wird an die `Output`-SIBs weitergegeben.

Entscheidung über die Ausgabe (grün)
Welche Ausgabe zurückgegeben wird, entscheiden die `If`-SIBs. Der Bezeichner beschreibt einen Pfadausdruck, der zu einem Wahrheitswert evaluiert wird. Das erste `If`-SIB erhält den Kontextknoten als Eingabe. Sie kann auch von allen folgenden `If`-SIBs verwendet werden. Der Pfadausdruck «`$input instance of element(paragraph)`» testet, ob der Kontextknoten ein `<paragraph>`-Element ist. In diesem Fall wird die `true`-Kante ausgewertet. Sie zeigt auf einen `DocumentNode`-SIB. In TT-Graphen beginnt die Definition einer Ausgabe immer mit einem SIB dieses Typs.

Wird der Pfadausdruck des obersten `If`-SIBs zu `false` evaluiert, wird dementsprechend der folgende `If`-SIB ausgewertet. Die nächsten beiden Entscheidungs-SIBs verarbeiten `<italic>`-Elemente und `<bold>`-Elemente. Handelt es sich bei dem Kontextknoten um keines dieser Elemente, evaluiert der Test des letzten `If`-SIBs zu `false`.

Deklaration der möglichen Ausgaben (orange):
Die `child`-Kante der `DocumentNode`-SIBs für die Verarbeitung von `<paragraph>`-, `<italic>`- und `<bold>`-Elemente zeigt jeweils auf einen `ElementNode`-SIB. Es ersetzt das Element durch ein `<p>`-, `<i>`- oder `<b>`-Element. Die `child`-Kante dieser SIBs zeigt auf einen `Output`-SIB. Die

Ausgabe besteht für ein `<paragraph>`-Element somit aus einem `<p>`-Element. Die Kinder werden durch die gleiche Template-Regel weiterverarbeitet. Die letzte Ausgabe wird ausgeführt, wenn das Kontextelement keines der drei vorher behandelten Elemente ist. In diesem Fall zeigt der `DocumentNode`-SIB direkt auf einen `Output`-SIB. So wird das unbekannte Element ignoriert und die Kinder werden weiterverarbeitet.

In diesem Graphen könnte mit GEAR geprüft werden, ob auf jeden `DocumentNode`-SIB irgendwann ein `Output`-SIB folgt, damit keine Ausgabe nur aus statischer Ausgabe besteht.

3.3.3 Tree-Transformer-SIBs

Das Tree-Transformer-Plugin hat eine feste Palette von SIBs für die Graphen der zweiten Ebene. Abbildung 3.7 zeigt die SIB-Icons mit dem zugehörigen Klassennamen der SIB-Klasse. Die UID entspricht, wie bei Graphen der untersten

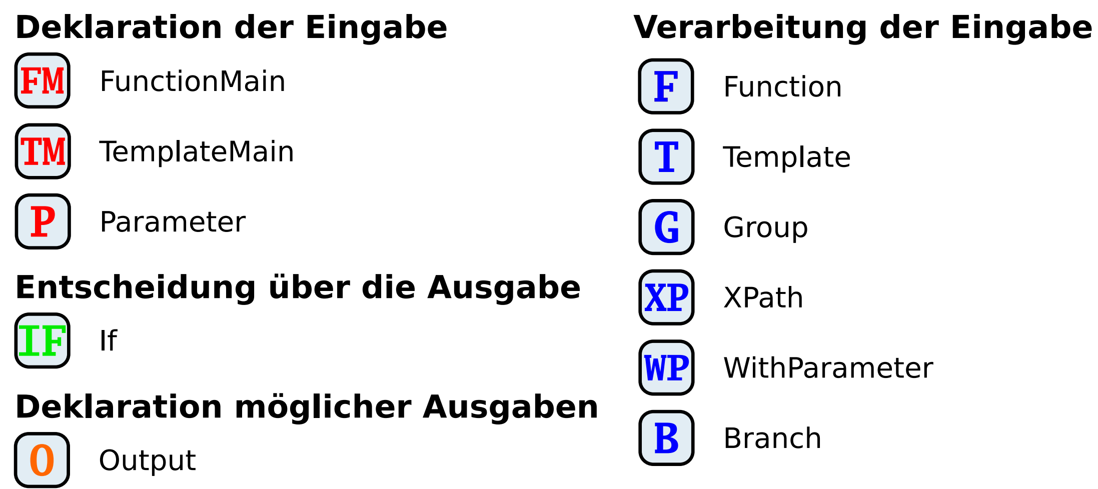

Abbildung 3.7: SIB-Icons der TT-SIBs

Ebene, dem vollqualifizierten Klassennamen der SIB-Klasse. Die Zugehörigkeit zu den TT-SIBs spiegelt sich im Java-Package der SIB-Klassen wider. Der vollqualifizierte Name des SIBs `TemplateMain` lautet wie folgt:

```
de.da.jn.sib.tt.TemplateMain
```

Die nachfolgenden Abschnitte geben einen detaillierten Einblick in die verschiedenen SIBs, ihre Parameter und den für sie definierten Achsen.

TemplateMain

Der `TemplateMain`-SIB ist der Haupt-SIB eines TT-Graphen, der einer Template-Regel entspricht. Er ist mit dem `DocumentNode`-SIB der XML-Graphen vergleichbar. Wie der `DocumentNode`-SIB hat er einen SIB-Parameter mit dem Namen `href`, der den relativen Speicherort des resultierenden Stylesheets festlegt. Der SIB-Parameter `pattern` enthält ein XSLT-Pattern. Es definiert die Knoten, für die die Regel verwendet wird. Der Modus der Template-Regel wird über den SIB-Parameter `modus` beschrieben. Letzterer wird für die Wiederverwendung in anderen Graphen standardmäßig als Modellparameter veröffentlicht. Der `TemplateMain`-SIB hat die `parameter`-Achse, über die die Parameter der Regel angebunden werden und die `result`-Achse, die den Kontextknoten für das nachfolgende SIB zur Verfügung stellt. Das Label der SIB-Instanz entspricht dem Wert des SIB-Parameters `pattern`. Lange XSLT-Pattern werden durch die Notation „. . . “ abgekürzt.

FunctionMain

Der `FunctionMain`-SIB ist ebenfalls ein Haupt-SIB eines TT-Graphen. Er wird statt des `TemplateMain`-SIBs verwendet, falls der Graph eine XSL-Funktion repräsentieren soll. Der qualifizierte Name der Funktion wird über den SIB-Parameter `name` gesetzt. Der Präfix und die URI des Namensraums werden ebenfalls über entsprechende SIB-Parameter gesetzt. Diese drei Parameter werden standardmässig als Modellparameter veröffentlicht. Weiterhin setzt der SIB-Parameter `href` den relativen Speicherort des generierten Stylesheets. Die `parameter`-Achse verbindet den `FunctionMain`-SIB mit seinen Parameterdeklarationen. Der Anzeigename entspricht dem `QName` der Funktion.

Parameter

Die Parameter der Haupt-SIBs werden über `Parameter`-SIBs definiert. Sie werden über die `sibling`-Achse miteinander verbunden. Ein Parameter wird über seinen Namen identifiziert. Der Name wird über den SIB-Parameter `name` gesetzt. Die Ergebnisachse stellt die Sequenz eines Parameters einem nachfolgenden SIB zur Verfügung. Die Anbindung einer Kette von Parametern erfolgt über die `parameter`-Achse ausgehend von dem entsprechenden `TemplateMain`- oder `FunctionMain`-SIB. Die Bezeichnung des SIBs entspricht dem Namen des Parameters.

Function

Eine XSL-Funktion, in Form eines TT- oder XSL-Graphen, wird mit dem speziellen Graph-SIB mit dem Namen `Function` in einen TT-Graphen eingebunden.

Die Ausgabesequenz des Funktionsaufrufs wird mit der `result`-Kante an den nachfolgenden SIB weitergegeben. Die `parameter`-Achse zeigt auf eine der Parameter, die wie die Parameterdefinitionen in einer Kette angeordnet sind. Es ist unerheblich, auf welche der Parameter die Kante zeigt. Der Name des referenzierten Modells dient als Anzeigename.

Template

Eine Template-Regel kann durch einen TT- oder XSL-Graphen repräsentiert werden. Der spezielle Graph-SIB `Template` referenziert den entsprechenden Transformationsgraphen. Die `group`-Achse verbindet die Regel mit einem `Group`-SIB. Wie bei `Function`-SIBs, wird der Name des Untergraphen als Bezeichner des SIBs verwendet.

WithParameter

Parameter werden als `WithParameter`-SIBs an `Group`- und `Function`-SIBs über die `parameter`-Achse gebunden. Die letzteren beiden SIBs definieren den Aufruf einer XSL-Funktion, beziehungsweise den Aufruf einer Gruppe von Template-Regeln. Mehrere Parameter werden mit einer `sibling`-Kante verbunden. Der Name des Parameters wird über den SIB-Parameter `name` gesetzt. Bei einem Aufruf einer Gruppe von Regeln oder einer Funktion muss der Name des `WithParameter`-SIBs mit dem entsprechenden Namen des Parameters übereinstimmen. In TT-Graphen wird dieser in den `Parameter`-SIBs festgelegt. Bei dem Aufruf von XSL-Graphen wird der Name eines Parameters in einer `<xsl:param>`-Deklaration gesetzt. Das Label einer `WithParameter`-Instanz stimmt mit dessen Namen überein.

Group

Das `Group`-SIB gruppiert mehrere `Template`-SIBs zu einer Gruppe. Die Zugehörigkeit zu einer Gruppe wird über den Modus der Template-Regeln definiert. Der SIB-Parameter `mode` legt den Namen des Modus fest. Der Bezeichner sollte eindeutig sein, um die Vermischung von Gruppen zu vermeiden. Die Eingabesequenz der Gruppe wird über eingehende Ergebniskanten erzeugt. Mehrere Eingabesequenzen werden zu einer verschmolzen. Für jeden Knoten der Sequenz wird nach einer Template-Regel aus der Gruppe gesucht, die den Knoten verarbeiten kann. Für Knoten, die keine Template-Regel behandelt, werden die built-in Template-Regeln von XSLT verwendet. Die Ausgabesequenzen der Regeln, die die Knoten verarbeitet haben, werden zu einer verschmolzen. Die `result`-Achse gibt die Ergebnissequenz der Template-Regeln an den nachfolgenden SIB weiter. Der Anzeigename des SIBs ist auf den Wert des Modus gesetzt.

XPath

Für einfache XPath-Ausdrücke, die mit den Standardfunktionen von XPath auskommen, steht der `XPath`-SIB zur Verfügung. Das Ergebnis des Ausdrucks wird über die `result`-Kante an das nachfolgende SIB weitergegeben. Auf die Eingabesequenz kann in dem Pfadausdruck über die Variable `$input` zugegriffen werden. Das Label von `XPath`-SIB-Instanzen entspricht dem Pfadausdruck. Lange Pfadausdrücke werden abgekürzt.

If

Verzweigungen entscheiden über die Ausgabe eines TT-Graphen. Ein `If`-SIB hat den SIB-Parameter `test`, der einen XPath-Ausdruck enthält. Dieser wird zu einem Wahrheitswert evaluiert. Die Variable `$input` ermöglicht den Zugriff auf die Eingabesequenz. Der SIB hat die `true`- und `false`-Achsen, die entweder auf einen weiteren `If`-SIB oder auf einen `DocumentNode`-SIB zeigen. Letzterer leitet die Deklaration der Ausgabe ein, die das Ergebnis des TT-Graphen darstellen, wenn die vorhergehenden `If`-SIBs den entsprechenden Wahrheitswert hatten.

Branch

Das `Branch`-SIB dient der Verteilung von Zwischenergebnissen. Eingehende Sequenzen von `result`-Kanten werden zu einer Sequenz verschmolzen. Ein `Branch`-SIB kann beliebig viele unbeschriftete, ausgehende Kanten haben. Die Nachfolger erhalten alle die vereinigte Sequenz als Eingabe. Die unbeschrifteten Kanten können mit `result`-Kanten kombiniert werden.

Output

Die Ausgabe wird neben Ergebniselementen, die durch XML-SIBs realisiert werden, durch `Output`-SIBs erzeugt. Ein `Output`-SIB schreibt die Eingabesequenz, die aus eingehenden `result`-Kanten und unbeschrifteten Kanten von `Branch`-SIBs zusammengesetzt sind, in die Ausgabesequenz.

3.3.4 Achsen

Die beschrifteten Kanten zwischen den Knoten in einem TT-Graphen heißen Achsen[2] (Axis). Im Folgenden werden die zusätzlichen Achsen beschrieben. Weiterhin sind die Kantenbeschriftungen der XSL-Graphen verfügbar, die vor allem für die Erzeugung der Ausgabe verwendet werden.

[2] Hier wird von der Bezeichnung „Branch“ aus dem LPC-Kontext abgewichen, da die beschrifteten Kanten in diesem Zusammenhang nicht einer Verzweigung, sondern einer Beziehung zwischen zwei Knoten entsprechen. Diese werden in XPath über Achsen realisiert.

result und unbeschriftete Kanten:
Die Ergebnisachse stellt die Ausgabesequenz eines TT-SIBs der nachfolgenden SIB-Instanz zur Verfügung. Zeigen mehrere Ergebniskanten auf ein SIB, werden die Sequenzen verschmolzen. Neben den unbeschrifteten Kanten eines `Branch`-SIBs zur Verteilung eines Ergebnisses an mehrere Nachfolger, ist dies der Kantentyp zur Weitergabe von Sequenzen. Beide Achsen werden daher unter dem Begriff Ergebniskante, beziehungsweise Ergebnisachse, zusammengefasst.

group:
Die Gruppierungsachse geht von einem `Template`-SIB aus und zeigt auf einen `Group`-SIB. Sie verbindet alle Template-Regeln, die auf den gleichen `Group`-SIB zeigen, über einen Modus.

parameter:
Der Modellierer bindet eine Kette von `WithParameter`-SIBs über die `parameter`-Achse an einen `Group`-SIB oder `Function`-SIB. Dadurch werden die Parameter in dem Aufruf der XSL-Funktion oder der Gruppe von Template-Regeln verwendet. Ferner wird bei der Deklaration einer Template-Regel oder XSL-Funktion ein `TemplateMain`- oder `FunctionMain`-SIB über die `parameter`-Achse an eine Kette von `Parameter`-SIBs gebunden.

true und false:
Diese Achse definiert eine Verzweigung. Sie wird von dem `If`-SIB verwendet.

3.3.5 Aufbau von TT-Graphen

Der Aufbau eines TT-Graphen lässt sich in vier Bereiche teilen, die klar voneinander getrennt sind:

1. Deklaration der Eingabe
2. Verarbeitung der Eingabe
3. Entscheidung, welche Ausgabe zurückgegeben werden soll
4. Deklaration der möglichen Ausgaben.

Abbildung 3.6 auf Seite 53 zeigt die Bereiche in einem Beispiel-Graph. In den nächsten Kapiteln wird näher auf diese Einteilung eingegangen.

Deklaration der Eingabe

Ein TT-Graph, der eine Template-Regel repräsentiert, beginnt mit einem `TemplateMain`-SIB. Wird er in eine XSL-Funktion übersetzt, startet er stattdessen mit einem `FunctionMain`-SIB. Die jeweilige SIB-Instanz wird als Start-SIB deklariert. Eine Template-Regel verarbeitet einen Knoten, den Kontextknoten. Er ist Teil der Eingabe. Beide Typen von TT-Graphen können optional Parameter deklarieren, die bei einem Aufruf der Funktion oder der Gruppe von Template-Regeln, der der aktuelle Graph angehört, übergeben werden. Die Parameter werden über die `parameter`-Achse des Haupt-SIBs angebunden. Es ist unerheblich, auf welchen der Parameter die Kante zeigt. Die Parameter werden durch `Parameter`-SIBs dargestellt und über die `sibling`-Achse verkettet. Die Abfolge der Parameter ist bedeutungslos, da der Name für die Zuordnung verwendet wird. Der Modellierer muss für jeden Parameter einen, für den Graphen eindeutigen, Namen angeben.

Die Parameter und der Kontextknoten bilden die Eingabe einer Template-Regel. Bei einer Funktion sind nur die Parameter Teil der Eingabe. Die Sequenzen werden über die `result`-Achse, der `TemplateMain`- und `Parameter`-SIBs, an nachfolgende SIBs weitergegeben.

Verarbeitung der Eingabe

Die Eingabe eines TT-Graphen wird von Gruppen von Template-Regeln, XSL-Funktionen und Pfadausdrücken weiterverarbeitet. Template-Regeln werden mit `Template`-SIBs referenziert. Die `group`-Achse verbindet eine Template-Regel oder mehrere Template-Regeln zu einer Gruppe. Die zu verarbeitenden Knoten werden der Gruppe zugeführt. Ferner werden Parameter als `WithParameter`-SIBs über ihre `parameter`-Achse an eine Gruppe gebunden. Die `sibling`-Achse verbindet sie zu einer Kette. Der Wert eines Parameters wird ebenfalls über eingehende Ergebniskanten bestimmt. Die Eingabe einer Template-Regel wird somit über das zugehörige `Group`-SIB festgelegt. XSL-Funktionen werden mit `Function`-SIBs aufgerufen. Die Parameter werden, wie bei dem `Group`-SIB über die `parameter`-Achse, an den `Function`-SIB gebunden. Es werden jedoch keine Ergebniskanten direkt an den SIB geleitet, da die Eingabe einer Funktion auf ihre Parameter beschränkt ist. Pfadausdrücke werden durch `XPath`-SIBs realisiert. Sie haben keine Parameter, sondern lediglich eine Eingabesequenz, die durch eingehende Ergebniskanten gebildet wird.

Template-Regeln, Funktionen und Pfadausdrücke erzeugen in TT-Graphen keine Ausgaben im Sequenzkonstruktor der Regel oder Funktion, die der Graph repräsentiert. Die Ausgabesequenzen werden als Variablen gehalten und können über die `result`-Kante an weiterverarbeitende SIBs gereicht werden. Endresultate werden an `Output`-SIBs weitergegeben. Diese werden in Kapitel 3.3.5

behandelt. Die Reihenfolge der verarbeitenden SIBs ist in XSLT unerheblich. Bei der Erzeugung von Zwischenergebnissen wird daher die Reihenfolge nicht modelliert. Lediglich die Ausgabe muss die Abfolge berücksichtigen.

Weiterhin können keine zwei oder mehr Zwischenergebnisse voneinander abhängen, da XSLT keine Assignmentregel hat und den einmal zugewiesenen Wert einer Variablen nicht mehr ändern kann. Diese Einschränkung verlangt, dass die Teilgraphen zur Verarbeitung der Eingabe kreisfrei sind. Bei der Generierung werden die Variablendeklarationen für Teilergebnisse topologisch sortiert. So sind alle abhängigen Variablen einer Variablendeklaration vorhanden.

In dem generierten Stylesheet eines TT-Graphen werden, losgelöst von der effektiven Ausgabe, alle Variablen für alle möglichen Ausgaben einer Transformation deklariert. Der verwendete XSLT-Prozessor berechnet nur die Variablen, die genutzt werden, so dass aus dieser Entscheidung kein Nachteil für die Performance der Transformation entsteht. Beispiel 6 zeigt eine Variable, die nicht verwendet wird.

Beispiel 6: Ungenutzte Variablen werden nicht evaluiert
Die Variable `$unused` wird nicht verwendet, da der Sequenzkonstruktor der `<xsl:if>`-Instruktion nie ausgeführt wird. Somit wird die Methode `tt:complexFunction()` nicht aufgerufen:

```
1 <xsl:variable
2     select="tt:complexFunction()"
3     name="unused"/>
4 <xsl:if
5     test="xs:boolean('false')">
6   <xsl:sequence
7       select="$unused"/>
8 </xsl:if>
```

Listing 3.3: Beispiel einer ungenutzten Variablen

Entscheidung über die Ausgabe

Ein TT-Graph kann unterschiedliche Ausgaben haben. Ist die Ausgabe eindeutig, beginnt sie mit einem `DocumentNode`-SIB. Der Graph enthält in diesem Fall keinen `If`-SIB. Die Ausgabe wird direkt in den Sequenzkonstruktor der Template-Regel oder der Funktion geschrieben. Andernfalls entscheiden `If`-SIBs, welche der möglichen Ausgaben von dem Sequenzkonstruktor des Graphen zurückgegeben werden. Ein `If`-SIB hat die zwei ausgehenden Achsen `true`

und `false`, die entweder auf einen weiteren `If`-SIB oder ein `DocumentNode`-SIB zeigen. Es dürfen keine Kreise entstehen und jeder Pfad, ausgehend von dem `If`-SIB mit Eingangsgrad 0 muss schließlich in einem `DocumentNode`-SIB enden. Bei der Generierung werden die `If`-SIBs topologisch sortiert und in `<xsl:choose>`-Instruktionen übersetzt. Die Blätter dieses binären Baums sind alle vom Typ `DocumentNode`. Wird ein `DocumentNode`-SIB über die `true`-Achse eines `If`-SIBs erreicht, schreibt der Generierungsprozess die Ausgabe in den Sequenzkonstruktor der `<xsl:when>`-Instruktion des `If`-SIBs. Sonst wird sie in die `<xsl:otherwise>`-Instruktion geschrieben.

Deklaration der möglichen Ausgaben

Die Ausgabe eines TT-Graphen beginnt mit einem `DocumentNode`-SIB. Die `child`-Achse zeigt auf die Ausgabesequenz. Sie besteht aus XML-SIBs, die als Ergebniselemente in den Sequenzkonstruktor eingebunden werden. Sie werden modelliert, wie ein XML-Graph. Weiterhin können `Output`-SIBs als Blätter in die Ausgabe eingefügt werden. Über die `sibling`-Achse kann ein Geschwisterknoten angebunden werden. Die Ausgabesequenz des `Output`-SIBs wird über eingehende Ergebniskanten gebildet und an die Stelle der Ausgabe des Sequenzkonstruktors geschrieben, an der er in dem Graphen positioniert ist. Auf diese Weise wird die gewünschte Reihenfolge der Ausgabe festgelegt.

3.3.6 Zusammenfassung

Die TT-Graphen abstrahieren von XSLT. Es werden lediglich oberflächliche XPath-Kenntnisse benötigt. Die Regeln von XSLT werden auf die Wesentlichen reduziert. Dies wird durch die Beschränkung auf wenige Kontrollstrukturen erreicht:

- Verzweigung
- Funktionsaufruf
- Aufruf einer Gruppe von Template-Regeln
- Auswertung eines Pfadausdrucks
- Erzeugung von Ausgabe durch XML-SIBs und dem `Output`-SIB.

Den TT-Graphen steht somit lediglich eine Untermenge der Funktionalitäten von XSLT zur Verfügung. Spezielle Graph-SIBs ermöglichen die Einbindung von weiteren TT-Graphen und XSL-Graphen. Auf diese Weise erschließt sich die gesamte Funktionalität von XSLT. Da XSLT keine Assignmentregel hat und

seiteneffektfrei ist, wird kein gemeinsamer Speicher benötigt. Weiterhin kommunizieren TT-Graphen nicht über vom Nutzer definierte Modellargumente mit in der Graphhierarchie darüberliegenden Graphen. Lediglich für die Generierung notwendige Informationen, wie der Name einer Funktion oder der Modus eines Templates, werden als Modellparameter exportiert. So ist die Wirkung eines TT-Graphen lokal und die Ausgabe allein durch seine Eingabe bestimmt. Weiterhin lässt er sich einfach wiederverwenden. Der Standard-Graph-SIB ermöglicht eine weitere Hierarchisierung von TT-Graphen, die eine Wiederverwendung von Graphfragmenten ermöglicht. Hier wird die Verwendung von Modellargumenten nicht eingeschränkt.

Das triviale Beispiel aus Kapitel 3.3.2 nutzt nicht die Vorteile der Einteilung in Komponenten, die durch einen XSLT-Experten implementiert werden können. Die einzige Template-Regel verwendet lediglich einen rekursiven Aufruf.

Die SIBs definieren nur die Achsen und Parameter, die für sie bestimmt sind. Fehlende Werte oder die falsche Anbindung von NachfolgeSIBs werden durch LocalChecker erkannt. Nicht alle Fehler können auf diese Weise gefunden werden, da die Tests lokal sind, dass heißt auf einen SIB und seine unmittelbaren Nachfolger beschränkt. Es lässt sich zum Beispiel konzeptionell nicht einfach prüfen, ob es nur einen Haupt-SIB in einem TT-Graphen gibt. Ferner kann GEAR Eigenschaften eines konkreten Modells sichern. Tree-Transformer vereinfacht den Entwurfsprozess, da viele Fehler frühzeitig erkannt werden.

3.4 XML/XSL-Graph

Ein XML- beziehungsweise XSL-Graph bildet ein XML-Dokument auf einen SIB-Graphen ab. Die Graphsyntax soll sich an dem XPath-2.0-Datenmodell (XDM) orientieren, damit sie für einen XSLT-Experten intuitiv ist. Das Konzept der Hierarchisierung ermöglicht wiederverwendbare und parameterisierbare XML-Fragmente. Ferner sollen weitere XSL-Graphen einfach eingebunden werden können.

Für die Modelllierung stehen sieben SIBs zur Verfügung, die die einzelnen Knotentypen darstellen, die es im XPath-2.0-Datenmodell gibt. Ein Dokumentknoten wird zum Beispiel durch einen `DocumentNode`-SIB repräsentiert. Im XDM ist die Wurzel eines Baums ein Dokumentknoten. Jeder XML-Graph enthält daher ein `DocumentNode`-SIB. An diesem erkennt das Plugin, dass es sich um ein XML-Modell handelt und den relativen Speicherort des Dokuments. Dort wird es gespeichert, wenn es in ein XML-Dokument übersetzt wird. Der Dokumentknoten hat genau ein Kind, das Wurzelelement, auch Dokumentelement genannt.

Die Baumstruktur eines XML-Dokuments wird eins zu eins in einem XML-Graphen abgebildet. Für die Verbindung zweier Knoten werden beschriftete

Kanten verwendet. Die Kanten entsprechen den einschrittigen Achsen[3] aus Pfadausdrücken.

In Abschnitt 3.4.1 werden die Verifikationsmethoden für XSL-Graphen dargestellt. Abschnitt 3.4.2 beschreibt ein Beispiel für einen XSL-Graphen. Die Abschnitte 3.4.4, 3.4.5 und 3.4.6 gehen näher auf die XML- und XSL-SIBs sowie auf die Kanten, beziehungsweise Achsen, ein.

3.4.1 Verifikationsmethoden

LocalChecker prüft während der Erstellung eines XSL-Graphen lokale Bedingungen an den SIB-Instanzen. Einige Beispiele hierfür sind:

- `<xsl:import>`-Deklarationen müssen vor allen anderen Top-Level-Deklarationen stehen. Daher wird gepüft, ob ein XSL-SIB vom Typ `Import` entweder einen vorhergehenden `Import`-SIB als Geschwisterknoten hat oder keinen.
- Ein `DocumentNode`-SIB ist die Wurzel eines XML-Baums. Daher darf er keinen Elternknoten besitzen.
- Ein wohlgeformtes XML-Dokument hat genau ein Dokumentelement. Das Kind des `DocumentNode`-SIBs darf somit keine Geschwisterknoten haben.
- Die `sibling`-Kante darf nur auf Elementknoten, Textknoten, Processing Instructions und Kommentare zeigen.
- Die `attribute`-Achse darf nur auf Attributknoten zeigen.

Wie bei TT-Graphen können globale Eigenschaften eines XSL-Graphen über GEAR gesichert werden.

3.4.2 Beispiel eines XSL-Graphen

Dieses Kapitel zeigt anhand eines Beispiels, wie ein XSL-Graph in XSLT übersetzt wird. Abbildung 3.8 stellt den Beispielgraphen dar. Listing 3.1 auf Seite 51 zeigt die Übersetzung in XSLT. Es handelt sich um das gleiche Beispiel wie in Kapitel „Beispiel eines TT-Graphen“ auf Seite 51. Die Transformation besteht aus drei Template-Regeln, die `<paragraph>`-, `<italic>`- oder `<bold>`-Elemente verarbeiten. Die erste Regel ersetzt `<paragraph>`-Elemente durch `<p>`-Elemente und ruft die `<xsl:apply-templates>`-Instruktion für die Kinder des Elements auf. Auf die gleiche Weise verfahren die beiden anderen Regeln. Sie ersetzen

[3] Mit einschrittigen Achsen sind diejenigen gemeint, die zwei Knoten direkt miteinander verbinden.

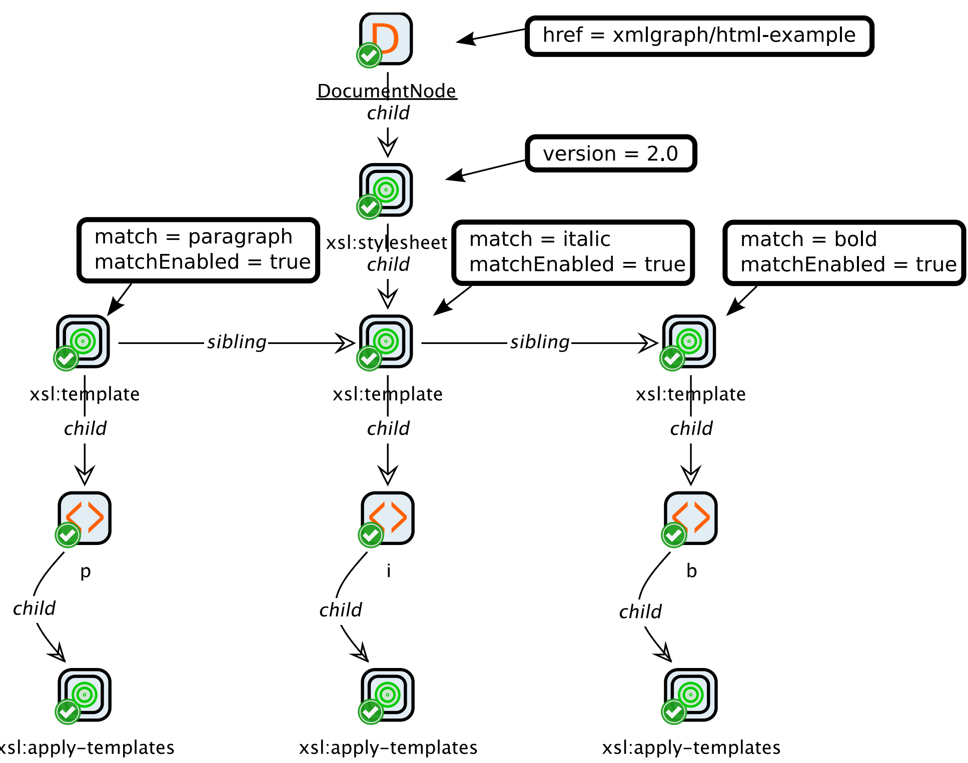

Abbildung 3.8: Beispiel eines XSL-Graphen

`<italic>`- oder `<bold>`-Elemente durch `<i>`- und `<b>`-Elemente. Textknoten werden durch die integrierte Regeln von XSLT unverändert in die Ausgabe kopiert.

In Abbildung 3.8 sind neben den SIBs die gesetzten Parameter mit ihren Werten aufgelistet. Parameter, die nicht verändert wurden, werden nicht angezeigt.

Die Wurzel des XML-Graphen ist ein `DocumentNode`-SIB. Der SIB-Parameter `href` des `DocumentNode`-SIBs legt fest, dass folgender Pfad als Speicherort dient: «`$deployFolder/xmlgraph/html-example.xsl`». Das Dokumentelement bildet der XSL-SIB `Stylesheet` mit dem Anzeigenamen «`xsl:stylesheet`». Dieser SIB repräsentiert ein `<xsl:stylesheet>`-Element. Der SIB-Parameter `version` hat den Wert «`2.0`». Der Wert wird in dem resultierenden Stylesheet dem Attribut `version` zugewiesen. Die Kinder des `Stylesheet`-SIBs, sind drei Top-Level-Deklarationen, die jeweils durch einen `Template`-SIB mit dem Anzeigenamen «`xsl:template`» repräsentiert werden. Sie stellen jeweils eine `<xsl:template>`-Deklaration dar. Für die `Template`-SIBs ist der SIB-Parameter `matchEnabled` auf `true` gesetzt, damit das `match`-Attribut in dem Stylesheet erscheint. Der Wert des Attributs ist «`paragraph`», «`italic`», beziehungsweise «`bold`». Die Kinder der `Template`-SIBs sind `ElementNode`-SIBs, die Ergebniselemente darstellen.

Sie ersetzen das Kontextelement durch `<p>`-, `<b>`- oder `<i>`-Elemente. Der Inhalt der `ElementNode`-SIBs besteht aus einem weiteren XSL-SIB, der die XSL-Instruktion `<xsl:apply-templates>` kapselt.

Für den Graphen in Abbildung 3.8 könnte mit GEAR die Bedingung gestellt werden, dass jede `<xsl:template>`-Deklaration eine `<xsl:apply-templates>`-Instruktion beinhalten muss. Wird der XSL-Graph um weitere Template-Regeln erweitert, markiert GEAR den Graphen erst als valide, wenn er eine `<xsl:apply-templates>`-Instruktion enthält, um die Kinderknoten des Kontextknoten zu verarbeiten.

3.4.3 Beispiel eines XSL-SIBs

Die XSL-SIBs repräsentieren ein XSLT-Konstrukt und sind in XSL-Graphen realisiert. Sie werden mit Graph-SIBs eingebunden. Der `Template`-SIB wird zum Beispiel durch den Graphen in Abbildung 3.9 repräsentiert. Er enthält einen

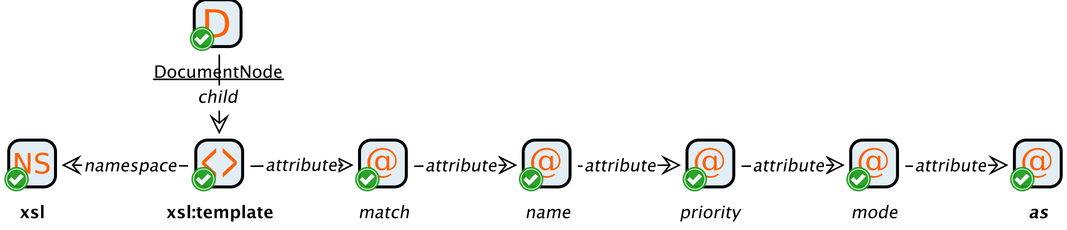

Abbildung 3.9: SIB Graph des XSL Template-Elements

Dokumentknoten, der auf den Elementknoten `<xsl:template>` verweist. Er hat den Anzeigenamen «`xsl:template`». Die Kindachse wird als Modellbranch exportiert. Dies wird im Graphen durch Fettschrift des SIB-Instanz-Namens visualisiert. Fünf `AttributeNode`-SIBs, deren Bezeichner dem Namen des Attributs entspricht, werden über die Attributachse an das Element geknüpft. Das ist eine Auswahl der Parameter, die ein `<xsl:template>`-Element haben darf. Die Attributknoten veröffentlichen den `value`- und `enabled`-Parameter. In dem darüberliegenden Graphen kann zum Beispiel das Attribut `match` über den SIB-Parameter `match` gesetzt werden. Damit das Attribut bei der Generierung in dem `<xsl:template>`-Element erscheint, muss ferner der SIB-Parameter `matchEnabled` auf `true` gesetzt werden. Des Weiteren wird von dem äußersten Knoten die Attributachse exportiert. Auf diese Weise können im darüberliegenden Graphen weitere Attribute an das Element geknüpft werden. Die Namensraumachse des SIBs, mit der Bezeichnung «`xsl:template`», führt zu einem `NamespaceNode`-SIB, das den Namensraum für den Präfix `xsl` auf den XSLT-Namensraum festlegt. Er veröffentlicht den `namespace`-Branch.

3.4.4 XML-SIBs

Ein XML-Graph ist die graphische Darstellung eines XML-Baums, wie es das XPath-2.0-Datenmodell vorschreibt. Jeder Knotentyp in dem XDM wird daher durch einen entsprechenden XML-SIB dargestellt. In Abbildung 3.10 sind die SIB-Icons mit dem zugehörigen Klassennamen der jeweiligen SIB-Klasse zu finden. Für eine einfache Transformation nach XSLT und eine einfache Modellierung werden redundante Zeiger zwischen Kind- und Elternelementen unterdrückt. Die Kindkante zeigt auf eines der Kinder. Geschwisterkanten verbinden die Kinder des Elements. Sie bilden eine Kette, die zugleich die Reihenfolge festlegt.

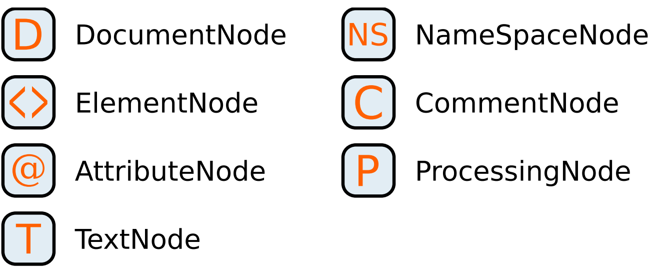

Abbildung 3.10: SIB-Icons der XML-SIBs

Die Zugehörigkeit eines SIBs zu den XML-SIBs kann an dem Java-Package abgelesen werden. Der vollqualifizierte Klassenname des SIB-Repräsentanten für Dokumentknoten lautet wie folgt:

```
de.da.jn.sib.xml.DocumentNode
```

Der eindeutige Bezeichner (die UID) eines XML-SIBs entspricht dem vollqualifizierten Namen der Klasse. Das erleichtert die Lesbarkeit von UIDs in XPath-Ausdrücken in Transformationen von SIB-Graphen gegenüber den automatisch generierten UIDs[4]. Für die Kapselung von XSLT-Fragmenten wird das Konzept der Hierarchisierung verwendet, das in jABC durch den Graph-SIB realisiert wird. Ein Beispiel hierfür sind die XSL-SIBs. Bei der Kapselung eines XML-Fragments dürfen Modellargumente veröffentlicht werden, da es eine Hierarchisierung innerhalb einer Komponente darstellt.

Die Komponente *Tree-Transformer-SIBs* (siehe Kapitel 4.1) verwaltet die Klassen der XML-SIBs. Die folgenden Abschnitte gehen detailliert auf die SIBs, die den Knotentypen in einem XML-Baum entsprechen, ein.

[4]Beispiel: «de426e7-a680-4402-838d-818ac9082aa2»

DocumentNode

Jeder XML-SIB-Graph muss genau einen Dokumentknoten enthalten. Tree-Transformer erkennt anhand dieser SIB-Instanz das Modell. Der `DocumentNode`-SIB hat lediglich die Kindachse, die auf das Dokumentelement vom Typ `ElementNode` zeigt. Es darf weder mehrere Dokumentknoten geben, noch darf ein Dokumentknoten mehrere Kinder haben. Werden diese Regeln missachtet, repräsentiert der SIB-Graph kein wohlgeformtes XML-Dokument. Diese und andere Regeln werden während der Modellierung durch LocalChecker-Regeln überprüft. Der einzige Parameter ist `href` vom Typ `String`. Er beschreibt den relativen Pfad des Dokuments, den dieses Modell repräsentiert. Für die Generierung einer Transformation speichert Tree-Transformer alle benötigten XSLT-Modelle als Dokumente in ein Verzeichnis. Der Parameter definiert den relativen Speicherpfad.

ElementNode

Elementknoten werden durch `ElementNode`-SIBs repräsentiert. Für sie sind alle Achsentypen definiert. Der Inhalt eines Elements wird durch die Kindachse angebunden. Kinder eines Elementes können Elemente, Text, Kommentare und Processing Instructions sein. Attribute werden in einer Kette über die Attributachse mit dem Element verbunden. Das Element ist der Startpunkt der Kette. Die `sibling`-Kante zeigt auf den in Dokumentreihenfolge nächsten Knoten. Der SIB hat die zwei SIB-Parameter `prefix` und `localName` vom Typ `String`. Zusammen bilden sie den lexikalischen `QName` des Elements. Beide Werte müssen `NCNames` sein. Als trennendes Zeichen wird der Doppelpunkt verwendet. Beispiel 7 zeigt einen lexikalischen `QName` mit und ohne Präfix.

Beispiel 7:
Der lexikalische `QName` zu dem Präfix „`xsl`“ und lokalen Namen „`template`“ hat die folgende Form:

```
xsl:template
```

Ist kein Präfix angegeben, wird der Doppelpunkt unterdrückt:

```
template
```

Eine SIB-Instanz vom Typ `ElementNode` erhält als Label den lexikalischen `QName`.

Der Präfix wird über ein `NamespaceNode`-SIB an einen Namensraum gebunden. Die Namensraumdeklarationen werden als Kette über die Namensraumachse an das Element gehängt.

AttributeNode

Attribute werden über die Attributachse verkettet. Ein `AttributeNode`-SIB definiert seinen qualifizierten Namen über die Parameter `prefix` und `localName`, wie bei den Elementknoten. Der Anzeigename entspricht dem lexikalischen `QName` des Knotens. Ferner sind die SIB-Parameter `value` vom Typ `String` und `enabled` vom Typ `boolean` für den Wert des Attributs bestimmend. Der `enabled`-Parameter steuert, ob ein Attribut gesetzt ist. Der Wert `false` hat den gleichen Effekt, als wäre der Attributknoten nicht an einen Elementknoten geknüpft. Bei der Hierarchisierung von XSL-Graphen können auf diese Weise optionale Attribute in einem Untermodell definiert werden und in der `GraphSIB`-Instanz, die auf das Submodell verweist, aktiviert beziehungsweise deaktiviert werden.

NamespaceNode

Namensraumdeklarationen haben nur einen Achsentyp für die Verkettung von `NamespaceNode`-SIBs. Ihr Anzeigename entspricht dem Präfix, für den sie definiert sind. Neben dem SIB-Parameter `prefix` vom Typ `String` für den Präfix, definiert der `uri`-Parameter eine Zeichenkette, die die URI des Namensraums repräsentiert.

TextNode

Ein Textknoten kann bei gemischtem Inhalt des Elternelements über die Geschwisterkante auf ein Element, einen Kommentar oder eine Processing-Instruction zeigen. XML-Parser erzeugen immer möglichst große Textknoten. Ein Textknoten kann nie einen weiteren Textknoten als Geschwisterknoten haben, da diese verschmolzen würden. Der einzige Parameter dieses SIB-Typs ist vom Typ `String`, heißt `text` und repräsentiert den Text des Knotens. Der Anzeigename eines Textknotens ist sein Text. Tree-Transformer kürzt lange Textinhalte ab und zeigt durch die Notation „...“ an, dass nicht der vollständige Inhalt angezeigt wird.

CommentNode

Ein Kommentar kann mit der `sibling`-Achse auf Text, ein Element, eine Processing-Instruction oder einen weiteren Kommentar verweisen. Der Inhalt wird über den SIB-Parameter `text` definiert. Das Label zeigt den Inhalt des Knotens, wie bei einem Textknoten.

ProcessingNode

Eine Processing Instruction hat die gleiche Schnittstelle wie ein Kommentar. Das `ProcessingNode`-SIB definiert seinen Inhalt über den `text`-Parameter und seinen Namen über den `name`-Parameter. Beide sind vom Typ `String`. Der Name der Processing-Instruction bildet den Anzeigenamen einer Instanz des SIBs.

3.4.5 XSL-SIBs

Mit den im vorhergehenden Kapitel vorgestellten SIBs können XSLT-Dokumente modelliert werden. Es wäre mühsam, sich wiederholende Fragmente, wie zum Beispiel die wichtigsten XSL-Befehle und Top-Level-Deklarationen, jedesmal neu zu erstellen. Die Kapselung in einen Graph-SIB ermöglicht die Wiederverwendung von Fragmenten ohne weiteren Modellierungsaufwand. Die gebräuchlichsten XSL-Anweisungen sind bereits im Plugin enthalten. Es ist möglich, vollständige XSL-Transformationen auf diese Weise zu kapseln. Letzteres ermöglicht auch das enthaltene Top-Level-Konstrukt `<xsl:import>`.

Jedes dieser SIB-Typen ist ein Graph-SIB und fest an ein Modell gekoppelt (*immutable Graph-SIB*). Sie kapseln genau ein Element aus der XSLT-Spezifikation. Jeder XSL-SIB exportiert die Kind-, Geschwister-, Attribut- und Namensraumachse als Modell-Branch. Sie dienen als Anknüpfpunkte für Erweiterungen. Eine Ergänzung kann in einen neuen Graph-SIB gekapselt und wiederverwendet werden.

In den Attributknoten wird der Parameter `value` exportiert. Der Modellparameter wird nach dem Attribut benannt. Ein obligatorisches Attribut hat den Parameter `enabled` auf `true` gesetzt und exportiert ihn nicht. Optionale Attribute werden durch Veröffentlichung des `enabled`-Parameters realisiert. Der Modellparameter erhält den Namen `enabledNAME`, wobei `NAME` dem Attributnamen entspricht. Lange Namen werden für beide Parameter in den Modellparametern abgekürzt.

Als Namensraum wird der XSLT-Namensraum verwendet. Dazu wird für den Präfix `xsl` die entsprechende URI gesetzt:

`http://www.w3.org/1999/XSL/Transform`

Diese Entscheidung zwingt einen Modellierer, sich an die Konventionen von XSLT zu halten.

Der Name einer XSL-SIB-Instanz entspricht in XSL-Graphen dem Namen des Modells, auf das es verweist. Der Name des Modells ist der qualifizierte Name des Elements, das es referenziert. Das Modell für das `<xsl:template>`-Element heißt dementsprechend `xsl:template`. Der Präfix schlägt sich im Java-Package der zugehörigen `GraphSIB`-Klasse nieder:

`de.da.jn.sib.xsl.Template`

Analog zu den XML-SIBs haben die XSL-SIBs als eindeutigen Bezeichner den vollqualifizierten Klassennamen der SIB-Klasse. Im weiteren Verlauf wird der `QName` eines Elements als Synonym für den entsprechenden SIB verwendet.

3.4.6 Achsen

Analog zu dem XPath-Datenmodell existieren in einem XSL-Graphen sogenannte Achsen. Sie setzen die Knoten eines XML-Baums zueinander in Beziehung. Pfadausdrücke nutzen sie zur Navigation im Baum. Sie werden in XML/XSL-Graphen durch beschriftete Kanten dargestellt. In der folgenden Auflistung wird die Bedeutung der einzelnen Achsen erklärt:

child:
Die Kindachse verbindet einen Elternknoten mit einem seiner Kinder. Die Geschwisterkanten identifizieren alle Kinder eines Elternknotens.

sibling:
Eine Geschwisterkante verbindet zwei Geschwisterknoten in Dokumentreihenfolge.

attribute:
Die Reihenfoge der Attribute ist nicht relevant und sie sind nicht Teil des Inhalts eines Elements. Daher gibt es eine Attributachse, die Attribute von anderen Kindelementen differenziert. Mehrere Attribute werden über die Attributachse verkettet.

namespace:
Namensraumdeklarationen haben ebenfalls eine eigene Achse und werden über diese verkettet, falls es für einen Knoten mehrere Deklarationen gibt.

XPath kennt weitere Achsen, die für die Selektion von Knoten verwendet werden. Für die möglichen Beziehungen zwischen zwei Knoten sind die oben genannten Kantentypen ausreichend.

3.4.7 Zusammenfassung

Ein XSL-Graph ist die kanonische Abbildung eines XML-Baums, wie es das XPath-2.0-Datenmodell vorschreibt. Die XML-SIBs repräsentieren die Knotentypen und die Achsen erlauben die Beziehungen zwischen Knoten, wie es das Datenmodell vorsieht. Der einzige Unterschied ist die Möglichkeit der Hierarchisierung innerhalb eines XSL-Graphen. Die Erstellung von XSL-Graphen ist somit für einen XSLT-Experten einfach zu erlernen.

Tree-Transformer kann ein XML-Modell in das repräsentierte XML-Dokument übersetzen oder ein bestehendes Dokument in ein Modell importieren. Ein XSLT-Experte ist somit nicht gezwungen, Stylesheets in XSL-Graphen zu erstellen. Ferner verwendet ein Modellierer XSL-Graphen, um Komponenten für TT-Graphen zur Verfügung zu stellen.

Für die XML-SIBs sind nur die Achsen und Parameter definiert, die das XDM vorschreibt. Auf diese Weise werden verschiedene Fehlerquellen beseitigt. Ein `AttributeNode`-SIB kann zum Beispiel keine Kinder haben, da für ihn die entsprechende Achse nicht definiert ist. Die Darstellung eines XML-Baums durch graphische Knoten verhindert, dass ein XSL-Graph nicht ausbalanciert ist, da es keine Start- und End-Tags wie in der Textform gibt, die die hierarchische Struktur zerstören könnten.

LocalChecker prüft Eigenschaften, die sicherstellen, dass der Graph ein wohlgeformtes XML-Dokument darstellt und teilt dem Anwender das Ergebnis mit. Die Attributachse darf zum Beispiel nur auf Attribute zeigen. Ferner wird für XSL-SIBs geprüft, ob ihre Position im Dokument und ihre Parameter der XSLT-Spezifikation entsprechen. Eine Auswahl dieser Tests ist:

- Das `<xsl:stylesheet>`-Element muss das Dokumentelement sein. Das bedeutet, es ist ein direktes Kind des `DocumentNode`-SIBs eines XSL-Graphen.
- Top-Level-Deklarationen müssen direkte Kinder von `<xsl:stylesheet>` sein.
- XSLT-Instruktionen müssen in einem Sequenzkonstruktor definiert sein. Das bedeutet, sie müssen direkte Kinder von einer XSLT-Deklaration sein, die einen Sequenzkonstruktor haben. Dazu zählen unter anderem `<xsl:template>`, `<xsl:function>` und `<xsl:variable>`.

Nicht alle Fehler kann LocalChecker abdecken, da nur lokale Bedingungen der SIBs und ihren direkten Vorgängern und Nachfolgern geprüft werden. Nicht erkannte Fehler werden schließlich durch den XSLT-Prozessor gefunden. Beispiele hierfür sind:

- Eine Variable muss deklariert werden, bevor sie verwendet wird.
- Ein Namensraum muss einem Präfix zugeordnet werden, bevor der Präfix verwendet werden kann.

Ferner kann GEAR globale Eigenschaften eines konkreten XSL-Graphen sichern.

Der Entwurfsprozess wird über die restriktive Modellierumgebung und durch die Prüfungen mit LocalChecker und GEAR vereinfacht, da Fehlerquellen früh erkannt werden können.

Die Einbindung von XSL-Graphen in einem XSL-Graphen wird über URIs der Form «xmlgraph://$modelid» realisiert. So können <xsl:import>-Deklarationen einen XSL-Graphen referenzieren. Tree-Transformer sorgt dafür, dass der entsprechende Graph in ein Stylesheet übersetzt und geladen wird.

3.5 Fazit

In allen drei Ebenen der Modellierung wird der Entwurfsprozess durch eine restriktive Modellierumgebung und Prüfungen mit LocalChecker sowie GEAR vereinfacht. Sie ersetzen jedoch nicht den XSLT-Prozessor für das Finden von Fehlern. Vor allem Laufzeitfehler, die sich durch falsche Annahmen über das Eingabedokument ergeben, sind während des Entwurfs schwer zu identifizieren. Abbildung 3.11 zeigt den vereinfachten Entwurfsprozess mit Tree-Transformer. Es muss nicht für jeden Fehler eine Übersetzung nach XSLT vorgenommen wer-

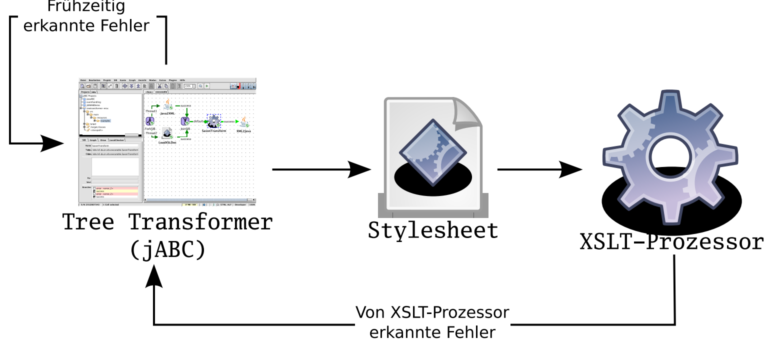

Abbildung 3.11: Vereinfachter Entwurfsprozess

den, um den Fehler durch den XSLT-Prozessor erkennen zu lassen. Stattdessen werden viele Fehler frühzeitig erkannt und können in jABC behoben werden. Das wird durch eine Kante, die bei dem Knoten beginnt, der jABC repräsentiert und auf sich selbst zeigt, visualisiert. In der Graphik ist nicht zu sehen, dass Tree-Transformer ermöglicht, mit einem Werkzeug die Anwendung, die eine Transformation ausführt, und die Transformation selbst zu modellieren.

Sowohl XSL- als auch TT-Graphen erreichen durch den Verzicht auf Modellargumente und den gemeinsamen Speicher eine lose Kopplung der Komponenten. Innerhalb einer Komponente wird ebenfalls das Konzept der Hierarchisierung verwendet. Da eine starke Kopplung innerhalb einer Komponente kein Problem darstellt, sind hier Modellargumente erlaubt, um gekapselte Fragmente von Transformationsgraphen zu parameterisieren. Dadurch ergibt sich eine flexiblere Wiederverwendbarkeit, als die von XSLT bereitgestellte Einbindung von Stylesheetmodulen.

Tree-Transformer ist nicht nur ein Editor-Ersatz. Weitere Einsatzfelder sind:

- jABC kann als Testumgebung für Anwendungen dienen, die Transformationsgraphen aufrufen. Tracer kann einen ausführbaren SIB-Graphen in jABC schrittweise ausführen und den gemeinsamen Speicher überwachen, um zu testen, ob eine Transformation funktioniert.

- Tree-Transformer kann als integrierte Entwicklungsumgebung für Anwendungen, die Baumtransformationen durchführen, eingesetzt werden. Genesys kann einen ausführbaren Graphen in eine Zielanwendung generieren. Tree-Transformer ermöglicht die Generierung der Transformationsgraphen in Stylesheets. Die Zielanwendung kann Transformationen durchführen ohne jABC zu benötigen.

- Das Plugin ist ein Werkzeug zur Aufgabenverteilung. Die Einteilung in drei Modellierungsebenen ermöglicht eine Trennung der Aufgabenbereiche nach Kompetenzen und Aufwand. Dadurch kann eine Steigerung der Effizienz erreicht werden.

- Transformationsgraphen dokumentieren eine Transformation graphisch. Weiterhin bietet jABC die Annotation von SIBs, Modellen und Projekten mit Dokumentation. Diese Informationen können dazu verwendet werden, ein Dokument zu generieren, ähnlich wie zum Beispiel Javadoc für die Sprache Java.

Das triviale Beispiel, dass in Kapitel 3.3.2 und 3.4.2 verwendet wird, zeigt nicht die Vorteile der Einteilung in wiederverwendbare Komponenten. Für das Tree-Transformer-Plugin müssen Transfomationsgraphen in Stylesheets übersetzt werden. Diese Transformationen werden in einem Bootstrapping-Vorgang wiederum mit Transformationsgraphen beschrieben. Dieses umfangreichere Beispiel liefert dahingehend bessere Ergebnisse. Sie werden in Kapitel 6 vorgestellt und bewertet.

Da XSLT keine generische Sprache wie Java ist, sondern auf Transformationen spezialisiert, müssen keine Komponenten von Drittanbietern als SIBs eingebunden werden. Daher werden für LDC keine SIB-Adapter und keine SIB-Klassen, außer den im Plugin enthaltenen, benötigt. So bleiben die SIB-Paletten für Baumtransformationen überschaubar.

4 Realisierung

Dieses Kapitel beschreibt die Realisierung von Tree-Transformer. Kapitel 4.1 stellt die Teilmodule, in die das Plugin unterteilt ist, vor. Kapitel 4.2 beschreibt das Bootstrapping, das Transformationsgraphen mit Tree-Transformer in Stylesheets übersetzt. Die Generierung von Transformationen wird in Kapitel 4.3 dargestellt. Den Import bestehender Stylesheets in XSL-Graphen beschreibt Kapitel 4.4. Auf die Implementierung der Java-Komponenten wird in Kapitel 4.5 eingegangen. Die Testmethoden für Tree-Transformer werden in Kapitel 4.6 dargelegt.

4.1 Systemaufbau

Das Tree-Transformer-Plugin ist in mehrere Teilmodule unterteilt. Die Verwaltung der Abhängigkeiten der Module und der Erstellungsprozess werden mit *Maven 2* [3] realisiert. Nachfolgend werden die einzelnen Module und ihre Bedeutung aufgelistet:

API:
In dem API-Modul werden die Interfaces für die Implementierungen im Framework- und Pure-Modul verwaltet. Ein Bootstrapmechanismus erlaubt die lose Kopplung der Schnittstellen an die Realisierung. Das ist nicht zu verwechseln mit dem Bootstrapping, das in Kapitel 4.2 beschrieben wird. Hierbei handelt es sich um eine Möglichkeit zur Einbindung einer Java-Klasse über die *Java-Reflection-API*, ohne die Klasse zu importieren. So kann zwischen mehreren Implementierungen gewählt werden. Es werden nicht die konkreten Klassen, sondern die Interfaces verwendet.

Plugin:
Das Plugin-Modul enthält die Plugin-Klasse, die als Schnittstelle zu jABC dient. Ferner bindet es die Funktionen zur Unterstützung der Modellierung in den jABC-Editor ein, die das Framework-Modul realisiert.

SIB:
Die XML-, XSL-, TT- und LPC-SIB-Paletten werden in dem SIB-Modul ver-

waltet. Dazu gehören auch der SIB-Adapter der LPC-SIB-Palette, die SIB-Graphen der XSL-SIBs[1], der LocalCheck-Code und die SIB-Icons.

Framework:
Das Framework enthält die Implementierung der Schnittstellen des API-Moduls. Dazu zählen zum Einen die Funktionen, die den Modellierer unterstützen. Zum Anderen realisiert das Framework die Komponenten der LPC-SIB-Palette. Die Funktionen werden über den zugehörigen SIB-Adapter aufgerufen. Die Interfaces werden jeweils für die einzelnen Bootstrapping-Schritte implementiert (siehe Kapitel 4.2). Ein SIB-Parameter der LPC-SIBs entscheidet, welche Implementierung ausgeführt wird.

Pure:
Die Klassen des Pure-Moduls implementieren, wie das Framework, die Interfaces für die LPC-SIB-Palette. Sie werden für generierte Transformationen verwendet, die unabhängig von jABC laufen. Die Bezeichnung «`Pure`» ist der Bezeichnung für `PureGenerator`-Generatoren von Genesys entlehnt, die Code erzeugen, der unabhängig von jABC läuft.

4.2 Bootstrapping

Die Transformationsgraphen von Tree-Transformer müssen nach XSLT übersetzt werden, damit ein XSLT-Prozessor sie ausführen kann. Dieser Vorgang lässt sich durch Baumtransformationen beschreiben. Das Plugin soll mit möglichst starkem Einsatz von jABC erstellt werden. Daher werden diese Transformationen in Transformationsgraphen beschrieben. Das Plugin erstellt somit seine Hauptfunktionalität aus sich selbst heraus.

Als Startpunkt dient ein manuell erstelltes Stylesheet, das im weiteren Verlauf Starttransformation genannt wird. Diese Transformation kann eine Untermenge der Transformationsgraphen nach XSLT übersetzen. Sie wird verwendet, um einen XSL-Graphen nach XSLT zu übersetzen, der eine größere Teilmenge übersetzen kann. Dieser Vorgang wird fortgeführt, bis die Transformationen die XSL- und TT-Graphen vollständig nach XSLT übersetzen können. Der gesamte Prozess wird in zwei Stufen durchgeführt, die jeweils aus mehreren Schritten bestehen. Die Modellierung der Transformationen in Transformationsgraphen dient der Konzeptvalidierung, weil die umfangreichen Transformationen der Graphen in XSLT viele Funktionen des Plugins verwenden.

Derartige Prozesse werden Bootstrapping genannt. Tree-Transformer modelliert den Kontrollfluss dieses Vorgangs in ausführbaren SIB-Graphen.

[1]XSL-SIBs sind immutable GraphSIBs.

Die Stylesheets, die die zweite Stufe des Bootstrappings erzeugt, können Transformationsgraphen nach XSLT übersetzen. Sie werden von dem Plugin verwendet, wenn in einem ausführbaren Graphen ein Transformationsgraph ausgeführt wird. Ein Beispiel hierfür ist in Abschnitt 3.2 auf Seite 49 beschrieben.

In Tabelle 4.1 werden einige graphische Elemente beschrieben, die in den Abbildungen 4.5 bis 4.9 verwendet werden. In den Abschnitten 4.2.1 und 4.2.2

Graphik	Beschreibung
SAXParser	Bezeichnet einen speziellen SAX-Parser, der SIB-Graphen nach XML serialisiert. Das Overlay-Icon oben links legt fest, dass es sich um eine manuell erstellte Komponente handelt.
TT TT-Graph-Level-2.1	Stellt einen SIB-Graphen im Speicher dar. Mit «`TT`» wird festgelegt, dass es sich um einen TT-Graphen handelt. Für XSL-Graphen wird stattdessen «`XSL`» eingesetzt.
XSL TT-Compiler-Level-1	Zeigt ein XSLT-Stylesheet. Das Overlay-Icon oben links zeichnet es als generiertes Dokument aus. XML-Dokumente, die kein Stylesheet sind, werden mit «`XML`» statt «`XSL`» gekennzeichnet.
import output	Die obere Kante stellt eine Eingabekante dar, die von außerhalb zugeführt wird. Die untere Kante zeigt aus einer Abbildung heraus. Sie stellt eine Ausgabekante dar, die an eine nachfolgende Komponente weitergereicht wird.
	Stellt einen XSLT-Prozessor dar. Die eingehende Kante mit der Bezeichnung «`style`» legt das Stylesheet fest. Die «`input`»- und «`output`»-Kante bestimmen die Herkunft des Eingabedokuments und den Bestimmungsort des Ausgabedokuments.

Tabelle 4.1: Beschreibung von Graphikelementen, die in folgenden Abbildungen verwendet werden

werden die beiden Stufen des Bootstrappings beschrieben. Dabei wird nicht auf die einzelnen Transformationsgraphen eingegangen, sondern die Ausführung des Bootstrappings im Überblick geschildert. In Abschnitt 4.2.3 wird dargestellt, wie die Stufen des Bootstrappings in einem ausführbaren Graphen integriert werden. Die Abschnitte 4.2.4, 4.2.5 und 4.2.6 gehen näher auf einzelne Komponenten, die daran beteiligt sind, ein. Auf eine detaillierte Darstellung der einzelnen Transformationen wird verzichtet, da sie den Rahmen dieser Arbeit sprengen würde. Abschnitt 4.2.7 fasst die Ergebnisse zusammen.

4.2.1 Bootstrapping-Stufe-1

Die Starttransformation kann eine Untermenge der XSL-Graphen nach XSLT übersetzen. Die erste Stufe des Bootstrappings erstellt TT-Compiler-Level-1, der TT-Graphen in Stylesheets umwandeln kann. So können in der zweiten Stufe beide Typen von Transformationsgraphen in Stylesheets transformiert werden.

Abbildung 4.1 zeigt den ausführbaren SIB-Graphen, der den Kontrollfluss von der ersten Stufe darstellt. Die SIB-Instanzen, in denen ein zentraler Teil des

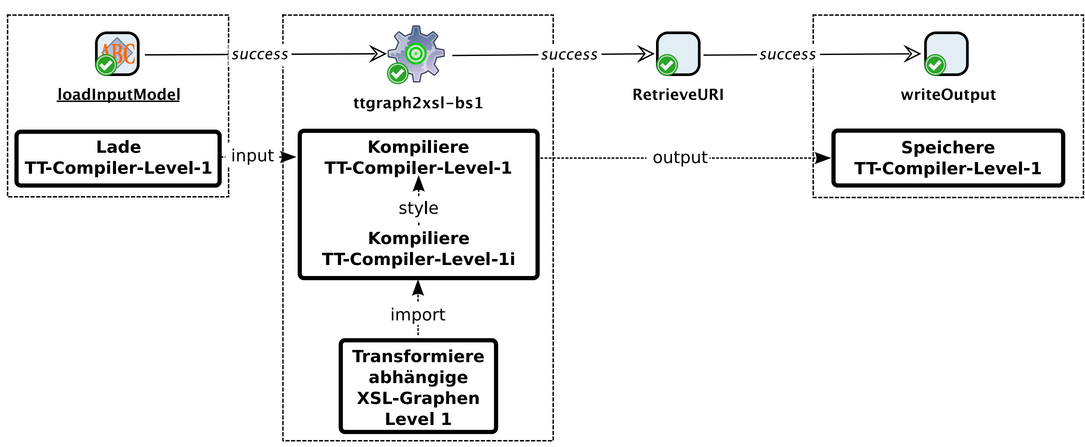

Abbildung 4.1: Bootstrapping-Stufe-1

Bootstrappings ausgeführt wird, sind in einen gestrichelten Rahmen eingefasst. Unterhalb dieser Instanzen sind jeweils schwarze Rechtecke angesiedelt, die eine kurze Beschreibung der Aufgabe enthalten, die dieses SIB erfüllt. Mit beschrifteten gestrichelten Kanten werden die Dokumente, die über den gemeinsamen Speicher zwischen den Komponenten ausgetauscht werden, gekennzeichnet. Die Bausteine werden in den Abschnitten 4.2.4, 4.2.5 und 4.2.6 näher erläutert. In diesem und dem nächsten Abschnitt soll ein Überblick über den Prozess vermittelt werden.

Die erste Stufe des Bootstrappings startet in einem `LoadTTModel`-SIB mit der Bezeichnung «`loadInputModel`». Er lädt die XML-Serialisierung eines TT-Graphen, der Graphen nach XSLT transformieren kann. Diese Transformation heißt TT-Compiler-Level-1. Das geladene XML-Dokument ist eine XML-Repräsentation des SIB-Graphen. Es wird als Eingabedokument für den folgenden `Transform`-SIB, mit dem Anzeigenamen «`ttgraph2xsl-bs1`», verwendet. Dieser kompiliert den TT-Compiler-Level-1 nach XSLT. Die Starttransformation transformiert hierfür, in einem ersten Teilschritt, einen XSL-Graphen nach XSLT, der seinerseits TT-Graphen nach XSLT übersetzen kann. Er wird TT-Compiler-Level-1i genannt, wobei «`i`» für „intermediate“ (Zwischenprodukt) steht. Er wird in einem zweiten Teilschritt verwendet,

um die XML-Serialisierung von TT-Compiler-Level-1 zu transformieren. TT-Compiler-Level-1i repräsentiert eine umfangreiche Transformation, die in mehrere XSL-Graphen aufgeteilt ist. Die importierten Graphen werden mit der Starttransformation zur Laufzeit in Stylesheetmodule übersetzt.

Ein `RetrieveURI`-SIB extrahiert den Speicherort des Ausgabedokuments aus dem TT-Graphen von TT-Compiler-Level-1. Die Ausgabe wird von einem `WriteTextFile`-SIB, mit dem Anzeigenamen «`writeOutput`», auf dem Dateisystem gespeichert. TT-Compiler-Level-1 steht der zweiten Stufe des Bootstrappings zur Verfügung.

4.2.2 Bootstrapping-Stufe-2

Die zweite Stufe des Bootstrappings erzeugt die Stylesheets, die in ausführbaren SIB-Graphen verwendet werden, um Transformationsgraphen auszuführen. Es stehen die Starttransformation, die XSL-Graphen nach XSLT übersetzt, und TT-Compiler-Level-1, für die Transformation von TT-Graphen in Stylesheets, zur Verfügung. Der Kontrollfluss wird in zwei ausführbare SIB-Graphen unterteilt, die jeweils ein Stylesheet erzeugen. Der in Abbildung 4.2 abgebildete Prozess erzeugt TT-Compiler-Level-2, der TT-Graphen nach XSLT übersetzt und TT-Compiler-Level-1 ersetzt. Ein analoger ausführbarer SIB-Graph erstellt

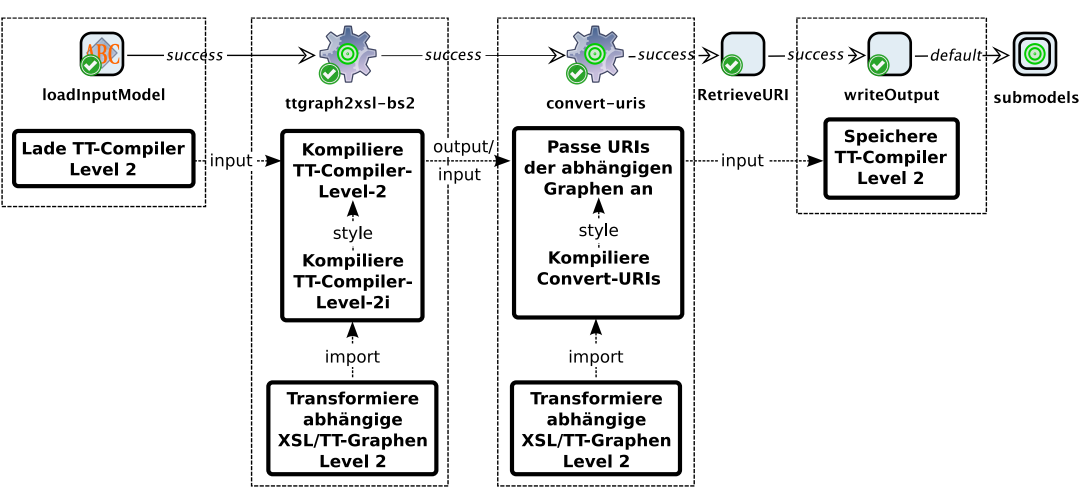

Abbildung 4.2: Bootstrapping-Stufe-2

XSL-Compiler-Level-2, der die Starttransformation ablöst.

Wie in Abbildung 4.1 werden SIBs, die eine relevante Aufgabe im Bootstrappingvorgang übernehmen, von einem gestrichelten Rechteck umrahmt, das zusätzlich eine kurze Beschreibung enthält. Der Kontrollfluss startet mit einem `LoadTTModel`-SIB mit dem Bezeichner «`loadInputModel`». Er lädt eine XML-Repräsentation des TT-Graphen, der TT-Compiler-Level-2 repräsentiert (siehe

Abschnitt 4.2.4). Dieser bildet das Eingabedokument für die Transformation des `Transform`-SIBs mit der Bezeichnung «`ttgraph2xsl-bs2`». Die Transformation ist in zwei Teile unterteilt. Zuerst wird der gleiche TT-Graph, der auch als Eingabedokument dient, mit TT-Compiler-Level-1 nach XSLT übersetzt. Das Ergebnis wird TT-Compiler-Level-2i genannt. Es wird verwendet, um anschließend das Eingabedokument in ein Stylesheet zu transformieren. Letzteres wird als TT-Compiler-Level-2 bezeichnet. Der gleiche TT-Graph wird demnach für den „intermediate" und den endgültigen TT-Compiler der zweiten Stufe verwendet, da dieser Graph TT-Graphen nach XSLT übersetzen kann und damit auch sich selbst. In Abschnitt 4.2.5 wird dieser Prozess genauer beschrieben. Sowohl TT-Compiler-Level-1 als auch TT-Compiler-Level-2i sind umfangreiche Transformationen, die Stylesheetmodule importieren. Ersterer verweist nur auf XSL-Graphen, während letzterer beide Graphtypen importiert. Diese werden zur Laufzeit geladen und mit der Starttransformation oder TT-Compiler-Level-1 übersetzt.

Im Unterschied zur ersten Stufe des Bootstrappings werden die Abhängigkeiten der Stylesheets, die in der zweiten Stufe erzeugt werden, nicht zur Laufzeit aus Transformationsgraphen erzeugt, sondern vorkompiliert. Bei der Ausführung von TT-Compiler-Level-2 oder XSL-Compiler-Level-2 liegen die abhängigen Stylesheetmodule auf dem Klassenpfad und werden direkt geladen. Zu diesem Zweck werden die Importdeklarationen der Stylesheets angepasst, so dass die Module vom Klassenpfad geladen werden.

Der nachfolgende `Transform`-SIB mit dem Anzeigenamen «`convert-uris`» erhält daher als Eingabe das Stylesheet TT-Compiler-Level-2. Für jedes importierte Stylesheet extrahiert er den Speicherort aus dem Haupt-SIB des zugehörigen Transformationsgraphen und ersetzt die URI. Diese Transformation wird mit dem TT-Graphen Convert-URIs durchgeführt. TT-Compiler-Level-1 konvertiert ihn in ein Stylesheet und ein XSLT-Prozessor führt ihn aus. Als Eingabe dient die Ausgabe der vorhergehenden Transformation.

Anschließend wird TT-Compiler-Level-2 mit angepassten Importdeklarationen auf dem Dateisystem gespeichert. Der `RetrieveURI`-SIB entnimmt hierfür dem Haupt-SIB des TT-Graphen den Speicherort.

Der letzte SIB ist ein Graph-SIB, der auf den Untergraphen in Abbildung 4.3 verweist.

In diesem SIB-Graphen werden die importierten Transformationsgraphen nach XSLT übersetzt, damit sie von TT- und XSL-Compiler-Level-2 verwendet werden können. Die Ausführung startet mit einem `Transform`-SIB mit dem Anzeigenamen «`all-imported-uris`». Er verwendet als Eingabe den TT-Graphen, der TT-Compiler-Level-2 repräsentiert, der bereits im aufrufenden SIB-Graphen, in dem Start-SIB geladen wurde. Für die Transformation wird der TT-Graph All-Imported-URIs verwendet. Dieser wird mit TT-Compiler-Level-1 nach XSLT übersetzt. Mit einem XSLT-Prozessor

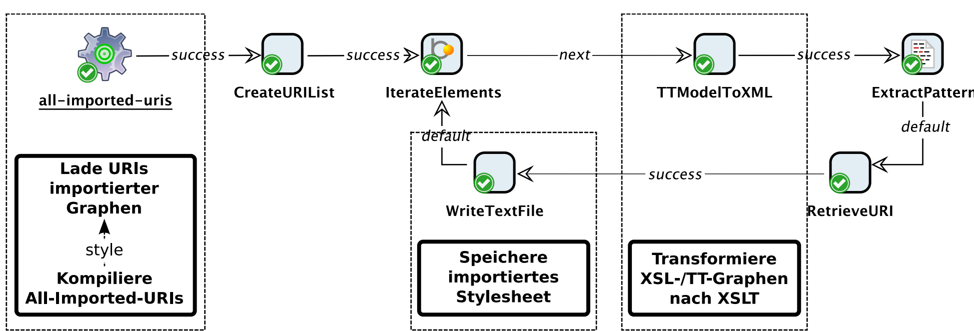

Abbildung 4.3: Generierung abhängiger Transformationsgraphen

ausgeführt, extrahiert er alle URIs von importierten Transformationsgraphen. Die Ausgabe ist eine Liste von URIs. Da ein Stylesheet als Ausgabe nur Text haben kann, wird in dem nachfolgenden `CreateURIList`-SIB eine *Java-List*, die Teil des *Java-Collections-Framework* ist, mit den URIs als Zeichenketten, daraus erzeugt.

Der `IterateElements`-SIB ist ein Common-SIB, der über eine Java-Liste iteriert. Solange die Liste nicht komplett abgearbeitet ist, wird der SIB über den `next`-Branch verlassen. Ansonsten wird der `exit`-Branch gewählt. In diesem Fall iteriert die SIB-Instanz über die Liste von URIs. Der `exit`-Branch ist als Modellbranch veröffentlicht und führt dazu, dass der Untergraph beendet und verlassen wird.

Für jede der importierten Transformationsgraphen wird die mit dem `next`-Branch beginnende Schleife ausgeführt. Sie startet in einem `TTModelToXML`-SIB, das den abhängigen Graphen nach XSLT transformiert. Handelt es sich um einen XSL-Graphen, wird dazu die Starttransformation verwendet. Für TT-Graphen wird TT-Compiler-Level-1 eingesetzt. Anschließend wird das importierte Stylesheetmodul auf dem Dateisystem gespeichert. Der Speicherort wird mit einem `RetrieveURI`-SIB aus dem Transformationsgraphen extrahiert. Dazu benötigt er die Model-ID des Graphen. Die verarbeiteten URIs haben die Form «`xmlgraph://$modelid`» oder «`ttgraph://$modelid`», je nachdem, ob es sich um einen XSL- oder TT-Graphen handelt. Sie enthalten somit die Model-ID. Der Common-SIB `ExtractPattern` führt ein Pattern-Matching mit einem regulären Ausdruck auf der aktuellen URI aus und extrahiert den eindeutigen Bezeichner für den `RetrieveURI`-SIB.

4.2.3 Integration der einzelnen Schritte

Die Schritte des Bootstrapping werden über einen ausführbaren Graphen referenziert und ausgeführt. Er ist in Abbildung 4.4 dargestellt. Der Graph verweist mit Graph-SIBs auf die entsprechenden ausführbaren Graphen. Über Modell-

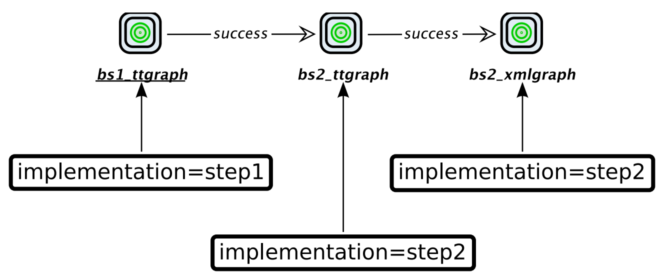

Abbildung 4.4: Aufruf der Schritte des Bootstrappings

parameter wird der zu verarbeitende Graph und die zu verwendende Implementierung in den Untergraphen gesetzt. Die Graph-SIBs mit der Bezeichnung «`bs2_xmlgraph`» und «`bs2_ttgraph`» referenzieren den gleichen ausführbaren Graphen, der in Abbildung 4.2 zu sehen ist.

4.2.4 Laden von Transformationsgraphen

Die `LoadTTModel`-SIBs des Bootstrappingvorgangs laden Transformationsgraphen für die Verwendung als Eingabedokument in einer XSLT-Transformation. Abbildung 4.5 zeigt die Schritte, die der SIB durchführt. Als Beispiel wird

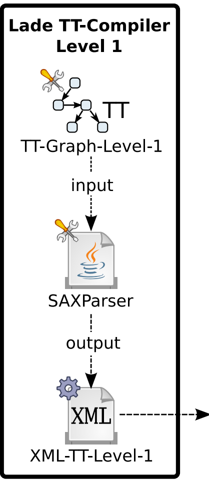

Abbildung 4.5: Laden des TT-Graphen für TT-Compiler-Level-1

der `LoadTTModel`-SIB der ersten Stufe des Bootstrappings aus Abbildung 4.1 verwendet. Der oberste Knoten entspricht dem TT-Graphen, der TT-Compiler-Level-1 repräsentiert. jABC stellt SIB-Graphen in einem Objekt vom Typ `SIBGraphModel` dar. Dieses wird als Eingabe für einen speziellen SAX-Parser verwendet, der das Objekt nach XML serialisiert. Der Parser wird durch den zweiten Knoten repräsentiert. Seine Ausgabe ist die XML-Serialisierung des

TT-Graphen, die der dritte Knoten darstellt. Er hat eine ausgehende Kante, die aus der Abbildung heraus zeigt. Das bedeutet, dass das XML-Dokument die Ausgabe dieser Komponente ist. In Abbildung 4.1 wird sie, ausgehend von dem Block mit der Bezeichnung «`Lade TT-Compiler-Level-1`», an die nachfolgende Transformation weitergegeben.

4.2.5 Ausführen von Transformationsgraphen

Transformationsgraphen werden über `Transform`-SIBs ausgeführt. Hierfür müssen sie zuerst in ein Stylesheet umgewandelt werden, um von einem XSLT-Prozessor interpretiert werden zu können. Dieser Prozess unterscheidet sich in den einzelnen Stufen des Bootstrappings. Anschließend wird das XSLT-Dokument mit einem XSLT-Prozessor ausgeführt, um das Eingabedokument zu transformieren.

Abbildung 4.6 zeigt, wie TT-Compiler-Level-1, in der ersten Stufe, erzeugt wird. Die Graphik entspricht dem Block unterhalb des `Transform`-SIBs mit dem Anzeigenamen «`ttgraph2xsl-bs1`» in Abbildung 4.1. Zu Beginn wird

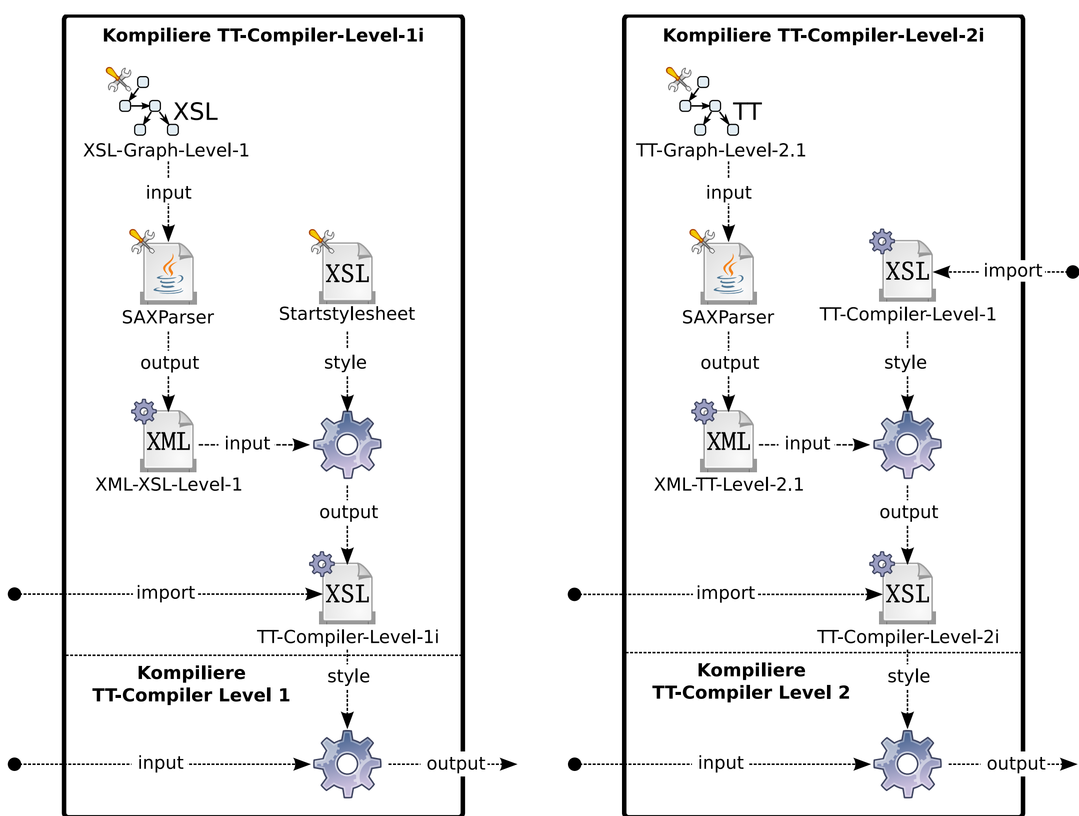

Abbildung 4.6: Erstellung des TT-Compiler-Level-1

Abbildung 4.7: Erstellung des TT-Compiler-Level-2

XSL-Graph-Level-1 mit einem SAX-Parser nach XML serialisiert. Das XML-Dokument wird als Eingabe für eine Transformation verwendet, die die Starttransformation als Stylesheet verwendet. Die Ausgabe ist TT-Compiler-Level-1i.

Der zweite Teil der Komponente ist durch eine gestrichelte Linie abgetrennt. Es wird das von außen zugeführte Eingabedokument mit TT-Compiler-Level-1i transformiert und der nächsten Komponente zur Verfügung gestellt. In diesem Fall handelt es sich bei der Eingabe um die XML-Serialisierung von TT-Graph-Level-1.

Die eingehende Kante mit der Bezeichnung «`import`», die auf TT-Compiler-Level-1i zeigt, visualisiert, dass dieses XSLT-Dokument Stylesheetmodule importiert. Abbildung 4.2.1 zeigt, dass diese durch XSL-Graphen repräsentiert werden und in einer weiteren Komponente zur Laufzeit nach XSLT transformiert werden. In Kapitel 4.2.6 wird dieser Vorgang beschrieben.

Abbildung 4.7 zeigt, wie in der zweiten Stufe TT-Compiler-Level-2 erstellt wird. Für die Transformation wird TT-Graph-Level-2.1 verwendet. Dabei handelt es sich um den gleichen Graphen, der TT-Compiler-Level-2 repräsentiert. Das ist möglich, weil TT-Graph-Level-2.1 TT-Graphen nach XSLT übersetzen kann und somit auch sich selbst. Er erhält die Bezeichnung mit der Endung «`2.1`», da XSL-Compiler-Level-2 ebenfalls durch einen TT-Graphen repräsentiert wird und den Namen TT-Graph-Level-2.2 erhält.

TT-Graph-Level-2.1 wird mit einem SAX-Parser nach XML serialisiert. TT-Compiler-Level-1 transformiert das Dokument nach XSLT. Er importiert XSL-Graphen, die zur Laufzeit geladen werden. Das daraus resultierende Stylesheet heißt TT-Compiler-Level-2i. Letzteres importiert TT- und XSL-Graphen. Der Unterschied zwischen einer Transformation in der ersten und der zweiten Stufe ist demnach, dass TT-Compiler-Level-1 zur Verfügung steht.

4.2.6 Umwandlung von Transformationsgraphen nach XSLT

Die Transformationsgraphen des Bootstrappings importieren Stylesheetmodule, die durch Transformationsgraphen repräsentiert werden. Weiterhin werden in dem ausführbaren SIB-Graphen in Abbildung 4.3 alle importierten Transformationsgraphen vorkompiliert. Hierfür müssen die Graphen nach XSLT übersetzt werden. Dies wird für XSL-Graphen in beiden Stufen durch die Starttransformation ermöglicht. TT-Graphen können erst ab der zweiten Stufe mit TT-Compiler-Level-1 transformiert werden.

In Abbildung 4.8 ist der Vorgang für XSL-Graphen und in Abbildung 4.9 für TT-Graphen zu sehen. Sie entsprechen jeweils dem Kompilieren eines Transformationsgraphen bei dessen Ausführung (siehe Kapitel 4.2.5). Das daraus resultierende Stylesheet wird nicht mit einem XSLT-Prozessor ausgeführt, sondern als Stylesheetmodul zurückgegeben.

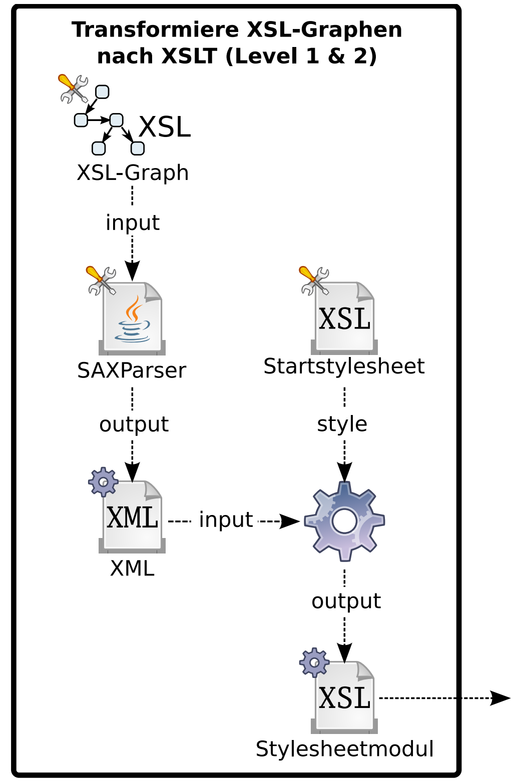

Abbildung 4.8: Umwandlung eines XSL-Graphen in ein Stylesheet (Level 1 und 2)

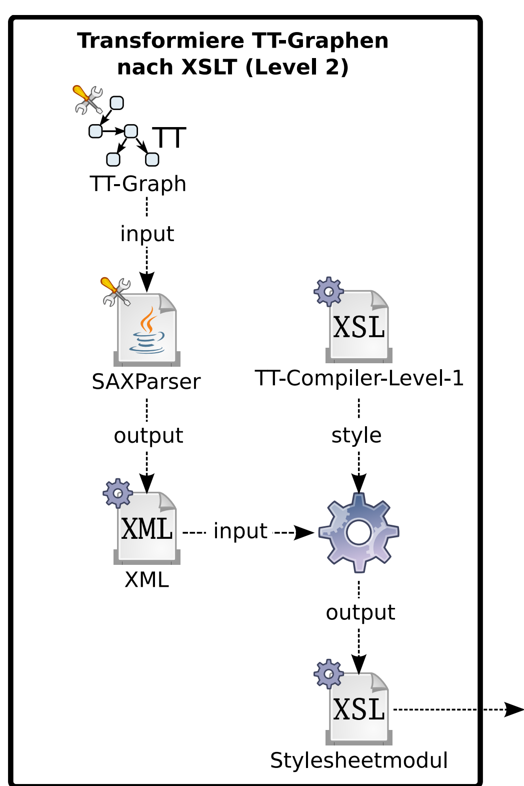

Abbildung 4.9: Umwandlung eines TT-Graphen in ein Stylesheet (Level 2)

4.2.7 Zusammenfassung

Die Transformationsgraphen, die in der zweiten Stufe des Bootstrappings erstellt werden, verwendet Tree-Transformer für die Übersetzung von Transformationsgraphen in Stylesheets. Die Hauptfunktionalität wird demnach vollständig mit den Mitteln des Plugins realisiert. Auch die Starttransformation, für die Transformation von XSL-Graphen nach XSLT, wird durch einen Transformationsgraphen ersetzt. Die Umsetzung mit Tree-Transformer zeigt, dass auch umfangreiche Transformationen mit diesem graphischen Ansatz beschrieben werden können. Einige Fehler im Konzept der TT-Graphen konnten während der Erstellung entdeckt und behoben werden. Weiterhin stellen die Transformationsgraphen einen umfangreichen Test der Funktionen des Plugins dar.

Im Folgenden werden einige Daten, die anhand des Bootstrappings gesammelt wurden, angegeben und ihre Bedeutung erläutert. TT-Compiler-Level-1 importiert 46 XSL-Graphen. Davon verwendet XSL-Compiler-Level-2 17 Graphen wieder und hängt von keinen weiteren Transformationskomponenten ab. TT-

Compiler-Level-2 nutzt 35 Transformationsgraphen der ersten Stufe des Bootstrappings und importiert 11 weitere. Demnach verwendet XSL-Compiler-Level-2 zu 100% und TT-Compiler-Level-2 zu 76% Transformationskomponenten der ersten Stufe.

4.3 Generierung von Transformationsgraphen

Mit Tree-Transformer erstellte Transformationsgraphen können in ausführbaren Graphen über den `Transform`-SIB aufgerufen werden. Die TT- und XSL-Graphen werden zur Laufzeit in Stylesheets übersetzt, um anschließend das Eingabedokument zu verarbeiten. Das Plugin Genesys ermöglicht einen ausführbaren Graphen in eine Anwendung zu übersetzen, die unabhängig von jABC ist. Die Transformationsgraphen sollten auch in generierten Anwendungen funktionieren.

Das jABC-Projekt und die zugehörigen Graphen werden bei der Generierung nicht kopiert und stehen der Anwendung daher nicht zur Verfügung. Weiterhin sollte die Anwendung ähnlich performant sein, als wäre sie ohne jABC erstellt worden. Die Umwandlung eines Transformationsgraphen in ein Stylesheet, zur Laufzeit, wäre daher unnötiger Mehraufwand.

Generiert ein Anwendungsexperte einen ausführbaren Graphen, der einen `Transform`-SIB enthält, in eine Anwendung, muss er zusätzlich den entsprechenden Transformationsgraphen in ein Stylesheet umwandeln. Das Stylesheet wird inklusive aller Stylesheetmodule im Vorhinein erstellt, damit sie zur Laufzeit vom Klassenpfad geladen werden können.

Die Vorkompilierung der XSL- und TT-Graphen ist in einem ausführbaren SIB-Graphen realisiert. Hierfür wird der gleiche Graph, wie in der zweiten Stufe des Bootstrappings verwendet. Statt XSL-Compiler-Level-2 und TT-Compiler-Level-2 wird der Transformationsgraph, der von dem `Transform`-SIB aus dem zu generierenden ausführbaren SIB-Graphen referenziert wird, übersetzt. In Abbildung 4.2 würde diese Änderung bedeuten, dass nicht der TT-Graph für TT-Compiler-Level-2 als Eingabe für die Transformation verwendet wird, sondern der entsprechende Transformationsgraph. Für die Transformation werden TT-Compiler-Level-2 und XSL-Compiler-2 verwendet.

Der Pfad des Hauptstylesheets muss angepasst werden, da die generierte Anwendung keine Information über den Pfad hat, an dem das Stylesheet bei der Erzeugung gespeichert wird. Die Anwendung hat als einzige Information die Model-ID des referenzierten Transformationsgraphen. Aus diesem Grund wird das Stylesheet nicht unter «`$deployFolder/$path`», sondern unter «`$deployFolder/${modelid}.xsl`» gespeichert. Die Variable `$deployFolder` hat den Wert des Verzeichnisses, in das die Stylesheets generiert werden.

Die entsprechenden Änderungen können mit Modellparametern realisiert wer-

den, so dass der Graph für den zweiten Schritt des Bootstrappings wiederverwendet werden kann.

Die `<xsl:import>`-Deklarationen des Hauptstylesheets enthalten URIs der Form `classpath://$path`. Sie zeigen auf die abhängigen Stylesheetmodule. In der generierten Anwendung wird für den `Transform`-SIB die `transform()`-Methode des `PureTransformer` (siehe Kapitel 4.5) ausgeführt. Sie lädt die XSLT-Dokumente von dem Klassenpfad. Die generierte Anwendung benötigt somit weder die Transformationsgraphen, noch die Implementierung des Tree-Transformer-Plugins. Erstere werden im Vorhinein in Stylesheets generiert. Letzteres wird durch die schlanke Implementierung des Pure-Projekts ersetzt.

4.4 Import von Stylesheets

Die Importfunktion von Tree-Transformer übersetzt XML in XML-Graphen. Das ermöglicht insbesondere den Import von XSLT in XSL-Graphen. Ein XSLT-Experte ist somit nicht gezwungen, jABC als Editor zu verwenden, um Komponenten für TT-Graphen zu erstellen.

Für den Import wird der neue XML-SIB `Element` eingeführt. Er entspricht einem XML-Element. Seine Attribute und Namespacedeklarationen werden als Parameter definiert. Auf diese Weise müssen diese nicht mit SIB-Instanzen dargestellt werden, wie bei der Verwendung des `ElementNode`-SIB. Tabelle 4.2 enthält die Parameter des SIBs, ihren Typ und jeweils eine kurze Beschreibung. Die Anbindung weiterer Namensraumdeklarationen und Attribute ist nicht vor-

Parameter	**Typ**	**Beschreibung**
`prefix`	`String`	Der Präfix des Elements
`localName`	`String`	Der lokale Name des Elements
`attributes`	`Map<String, String>`	Die Attribute des Elements. Jedes Attribut wird als ein Eintrag der `Map` gesetzt. Der Schlüssel entspricht dem Namen und der Wert dem Wert des Attributes.
`namespaces`	`Map<String, String>`	Die Namespace-Knoten, die für dieses Element definiert sind. Eine Namensraumdeklaration wird als Eintrag in der `Map` gesetzt. Der Schlüssel entspricht dem Präfix und der Wert der Namespace-URI.

Tabelle 4.2: Parameter `Element`-SIB

gesehen. Der SIB hat daher nur die Achsen `child` und `sibling`.

Die Übersetzung in einen SIB-Graphen wird mit einem *StAX-Parser* (**St**reaming **A**PI for **X**ML [11]) realisiert. Zu Beginn des Imports wird eine neue `SIBGraphModel`-Instanz erzeugt. Dieser Typ wird von jABC verwendet, um SIB-Graphen im Speicher darzustellen. Der Parser liest ein XML-Dokument ein und erzeugt für die entsprechenden Knoten SIB-Instanzen, die über Methoden des `SIBGraphModel`-Objekts in den SIB-Graphen eingefügt werden. Die Dokumentwurzel wird durch ein `DocumentNode`-SIB repräsentiert. Elemente und ihre Attribute sowie Namensraumdeklarationen werden mit einen `Element`-SIB dargestellt. Text, Kommentare und Processing-Instructions werden durch Ihre Gegenstücke in den XML-SIBs ersetzt. Bei diesem Prozess werden auch die Kanten mit der jeweiligen Achsenbeschriftung zwischen den SIB-Instanzen erzeugt. Hierfür bietet die Klasse `SIBGraphModel` ebenfalls Methoden. Nach dem Import kann mit einem der Layouter von jABC die Position der Knoten angepasst werden, so dass der Graph lesbar ist.

4.5 Implementierung in Java

Die Funktionalitäten des Tree-Transformer-Plugins sind in Java und mit jABC realisiert. Dieses Kapitel befasst sich mit der Implementierung der Java Komponenten. Diese lassen sich grob in vier Bereiche einteilen:

- Anbindung des Plugins an jABC
- Funktionen zur Unterstützung der Modellierung
- SIB-Klassen
- Komponenten für die ausführbaren SIBs

Für die Implementierung der Funktionen zur Unterstützung der Modellierung und die Komponenten für die ausführbaren SIBs stellt das API-Modul jeweils eine *abstract Factory* zur Verfügung, die über die Java-Reflection-API ihre Implementierung lädt. Abstract Factory ist ein Software Entwurfsmuster, das ermöglicht, mehrere Implementierungen der gleichen API zu haben. Bei der Verwendung der Funktionalität werden die Interfaces verwendet und nicht die konkreten Klassen. Die Implementierung kann ohne Anpassung ausgetauscht werden. Die Verwendung dieses Patterns dient zwei Zwecken. Einerseits lässt sich die Implementierung des Plugins austauschen, andererseits wird die gleiche abstract Factory von dem Tree-Transformer-Plugin mehrfach implementiert, da sie in unterschiedlichen Kontexten verwendet werden.

Die Factories werden über die abstrakten Klassen `TTPluginFactory` (Plugin-Factory) und `TTSIBFactory` (SIB-Factory) erstellt. Sie besitzen jeweils die überladene Methode `newInstance()`, um ein konkretes Objekt der abstrakten Klasse

zu erhalten. Wird die Methode parameterlos aufgerufen, liest die Factory eine *Java-Property* aus. Als Schlüssel verwendet sie den vollqualifizierten Klassennamen der Factory-Klasse. Der Wert der Property wird als der Name der konkreten Klasse verwendet. Ist die Property nicht gesetzt, lädt die Factory die erste Ressource auf dem Pfad: «`META-INF/services/$factory-class-name`». Die Datei beinhaltet den Namen der zu ladenden Klasse. Alternativ kann der Programmierer einen `String`-Parameter angeben, der dem vollqualifizierten Namen der implementierenden Klasse entspricht. Abbildung 4.10 zeigt ein vereinfachtes UML-Klassendiagramm der Factory-Klassen.

TTPluginFactory
+PROPERTY : String
+newInstance() : TTPluginFactory +newInstance(impl : String) : TTPluginFactory +addModelListener() : TTPluginFactory

TTSIBFactory
+PROPERTY : String
+newInstance() : TTSIBFactory +newInstance(impl : String) : TTSIBFactory +newTransformer() : Transformer

Abbildung 4.10: UML-Klassendiagramm der Factory-Klassen

Transformationen werden im Tree-Transformer-Plugin mit dem XSLT-Prozessor Saxon [13] ausgeführt. Für die Ansteuerung wird die JAXP-API [11] verwendet.

Die nachfolgenden Kapitel beschreiben die oben genannten Teilbereiche des Plugins.

4.5.1 Anbindung des Plugins an jABC

Die Klasse `TTPlugin` im Plugin-Modul implementiert das `Plugin`-Interface des jABC-Frameworks. Sie meldet den Tree-Transformer im jABC-Editor an und ruft, bei der Initialisierung des Plugins, die Pluginfunktionen über die `TTPluginFactory` auf.

4.5.2 Funktionen zur Unterstützung der Modellierung

Die Plugin-Factory kapselt die Funktionen des Plugins zur Unterstützung der Modellierung in jABC. Derzeit definiert sie lediglich die abstrakte Methode `addModelListener()`. Sie wird von der konkreten Factory `FrameworkPluginFactory` implementiert. Die `TTPlugin`-Klasse holt sich bei der Initialisierung des Plugins ein Objekt der Factory und ruft die `addModelListener()`-Methode auf. Diese erzeugt einen `TTModelListener` und meldet ihn bei dem jABC-Editor an. Die Klasse implementiert das Interface `ModelListener`. Es definiert mehrere Event-Methoden, die bei dem jeweiligen

Ereignis auf dem aktuellen SIB-Graphen aufgerufen werden. So kann das Plugin auf Veränderungen des aktiven SIB-Graphen reagieren. Die Implementierung der Factory stellt das Plugin-Framework. Werden bei der Weiterentwicklung von Tree-Transformer Funktionen hinzugefügt, die nicht über Ereignisse an SIB-Graphen angestoßen werden, kann die Plugin-Factory um weitere Methoden erweitert werden. Abbildung 4.11 zeigt ein UML-Klassendiagramm der Klassen `TTPluginFactory`, `FameworkPluginFactory` und `TTModelListener`.

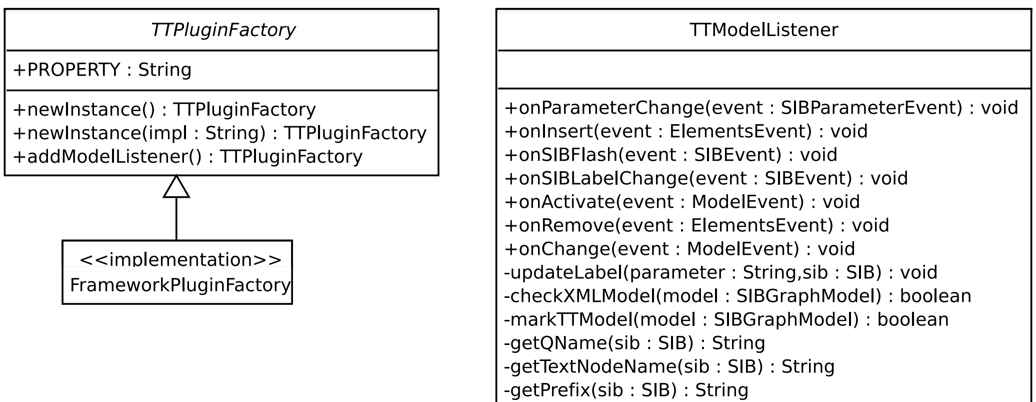

Abbildung 4.11: UML-Klassendiagramm der `TTPluginFactory` und ihrer Implementierung

4.5.3 SIB-Klassen

Die SIB-Klassen bestehen größtenteils aus generiertem Code, der mit dem SIBCreator-Plugin erstellt wurde. Nachträglich ist LocalChecker-Code hinzugekommen, der lokale Bedingungen an den SIBs prüft.

Die ausführbaren SIBs `Transform`, `LoadTTModel` und `TTModelToXML` sind nach dem SIB-Adapter-Entwurfsmuster entworfen. Sie implementieren die Interfaces des Tracer-Plugins, für die Extruder-Generierung und den Java-Class-Generator. Der `TTSIBAdapter` ist der SIB-Adapter für die drei SIBs. Er holt über die SIB-Factory ein konkretes Objekt des `Transformer`-Interfaces. Anschließend ruft er auf der Instanz die Methode auf, die die Funktionalität des SIBs repräsentiert. Der Modellierer wählt mit dem SIB-Parameter `implementation`, welche Implementierung der SIB-Factory verwendet werden soll. Für das Bootstrapping kann die Implementierung der jeweiligen Stufe angegeben werden (Stufe eins oder zwei beziehungsweise «`bs1`» oder «`bs2`»).

Ein Anwender des Plugins benötigt nur die Standardeinstellung (Stufe drei beziehungsweise «`default`»). Der SIB-Adapter gibt bei dem Aufruf der

`newInstance()`-Methode der SIB-Factory den Klassennamen der Implementierung an, die der Modellierer gewählt hat. Abbildung 4.12 zeigt die Beziehungen zwischen den SIB-Klassen, dem SIB-Adapter und der SIB-Factory. Weder der

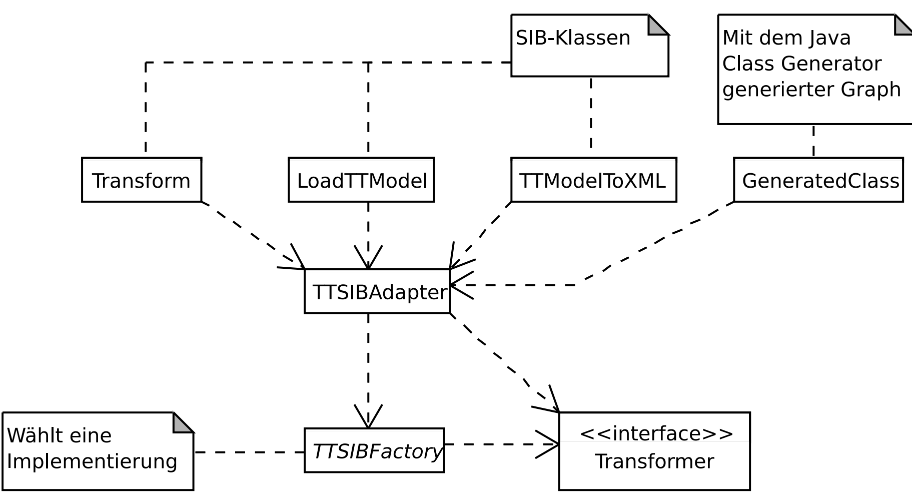

Abbildung 4.12: UML-Klassendiagramm, das die lose Kopplung durch das SIB-Adapter-Entwurfsmuster zwischen ausführbaren SIBs und ihren Komponenten zeigt

SIB-Adapter noch die SIBs haben eine direkte Abhängigkeit zu der verwendeten Implementierung. Die Klasse `GeneratedClass` in Abbildung 4.12 stellt eine mit dem Java-Class-Generator generierte Anwendung dar. Der übersetzte SIB-Graph führt über das `Transform`-SIB einen TT-Graphen aus. Die Abbildung zeigt, dass die Anwendung Methoden direkt auf dem SIB-Adapter aufruft. Die SIBs werden nicht benötigt. Jede Methode im `TTSIBAdapter` ist überladen. Die generierte Anwendung ruft jeweils die Methode ohne Angabe des Klassennamens der Implementierung der SIB-Factory auf. Der SIB-Adapter holt in diesem Fall die `PureSIBFactory`, die das Pure-Modul enthält. Die Wahl einer Factory in dem SIB-Parameter `implementation` hat somit keinen Einfluss auf die Ausführung in einer generierten Anwendung.

Die ausführbaren SIBs `CreateURIList` und `RetrieveURI` werden nur für das Bootstrapping verwendet. Sie können mit Tracer ausgeführt und mit Extruder generiert werden. Die Implementierung hängt nur von dem jABC-Framework und Klassen aus dem Java-Runtime-Environment ab. Weiterhin werden sie nur innerhalb des Plugins verwendet. Aus diesen Gründen steht der Code zur Ausführung direkt in der `trace()`-Methode, die Tracer verwendet. Die `execute()`-Methode für Extruder wird über einen Delegationsaufruf auf die `trace()`-Methode realisiert.

Das SIB-Modul bietet weitere SIB-Paletten, die aus nicht ausführbaren SIBs bestehen. Dazu zählen die XML-, XSL- und TT-SIBs. Sie werden in XSL- und TT-Graphen verwendet. Diese Graphen werden mit den Transformationen, die während des Bootstrappings erstellt werden, in XSLT-Stylesheets transformiert. Die Übersetzung eines SIBs ist nicht an die SIB-Klasse gekoppelt, sondern in den Transformationsprozess integriert. Dies ist möglich, da die SIB-Paletten nicht ständig erweitert werden und die Stylesheetmodule einer Transformation als weitere Graphen und nicht über neue SIB-Klassen integriert werden. Diese SIBs benötigen somit nur die Deklarationen der Achsen und SIB-Parameter, Dokumentation, ein Icon, eine UID und LocalCheck-Code. Eine Ausnahme bilden die XSL-SIBs. Sie sind immutable GraphSIBs und verweisen auf einen XSL-Graphen, der das jeweilige XSLT-Konstrukt, als XSLT-Fragment, modelliert.

4.5.4 Komponenten für die ausführbaren SIBs

Die Komponenten für die ausführbaren SIBs stellen die konkreten SIB-Factories zur Verfügung. Ihre Implementierungen bestehen vor allem aus folgenden Klassen:

- Factory-Klasse
- `Transformer`-Klasse (Transformer)
- `URIResolver`-Klasse (URI-Resolver)

Eine Instanz einer konkreten Factory-Klasse wird von der abstrakten Klasse `TTSIBFactory` mit der Methode `newInstance()` erzeugt. Als Parameter wird der Klassenname der konkreten Klasse angegeben. In Abbildung 4.13 sind die Spezialisierungen der abstract Factory und das `Transformer`-Interface zu sehen.

Die Instanz der konkreten Factory-Klasse erstellt mit `newTransformer()` ein Objekt der zugehörigen Implementierung des `Transformer`-Interfaces. Die `BS1Factory` erzeugt zum Beispiel einen `BS1Transformer`. Der Rückgabetyp der Methode ist der allgemeine Typ `Transformer`, der die jeweilige Implementierung versteckt.

Die Realisierung der Funktionen der ausführbaren SIBs wird über das `Transformer`-Interface angesprochen. Es hat für den `Transform`-, `LoadTTModel`- und `TTModelToXML`-SIB jeweils eine Methode, die seine Funktionalität kapselt. Das Interface wird für die einzelnen Schritte des Bootstrapping und für mit dem Java-Class-Generator generierte Anwendungen implementiert. So können für jeden Schritt des Bootstrappings, für die Ausführung von Transformationsgraphen und generierte Anwendungen, die gleichen SIBs verwendet werden.

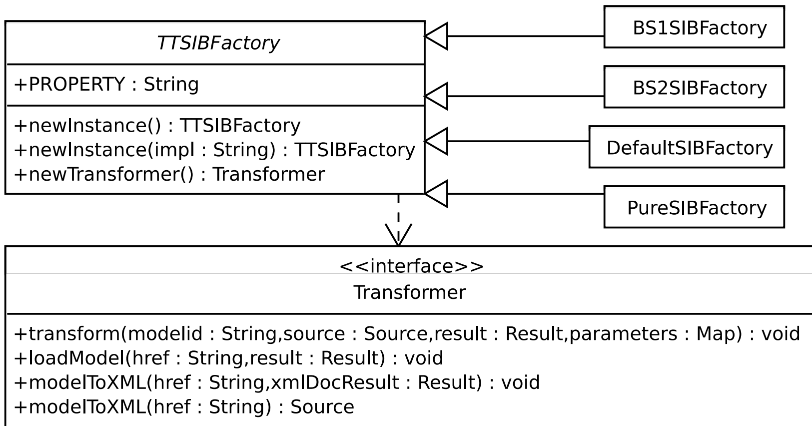

Abbildung 4.13: UML-Klassendiagramm der `TTSIBFactory`, ihren konkreten Klassen und des `Transformer`-Interfaces

Der Abbildung 4.14 ist zu entnehmen, dass `BS2Transformer` eine Spezialisierung von `BS1Transformer` ist und `DefaultTransformer` wiederum `BS2Transformer` erweitert. Der `BS1Transformer` implementiert die Methoden des `Transformer`-Interfaces. Die beiden abgeleiteten Klassen verwenden den

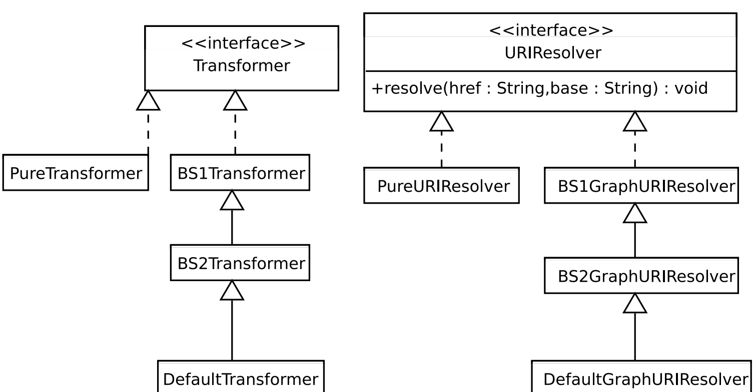

Abbildung 4.14: UML-Klassendiagramm der `Transformer`- und `URIResolver`-Realisierungen

Code der Basisklasse und überschreiben einige Methoden, die `protected` sind.

Diese drei Transformer sind Teil des Framework-Moduls und können über den SIB-Parameter `implementation` in den ausführbaren SIBs gewählt werden. Führt Tracer oder eine mit Extruder generierte Anwendung den SIB aus, lädt der `TTSIBAdapter` den entsprechenden Transformer. Mit Java-Class-Generator

generierte Anwendungen sind eine Ausnahme und werden später behandelt.

Der `Transform`-SIB ist der ausführbare SIB, der am häufigsten außerhalb des Plugins verwendet wird. Er ist ein spezieller Graph-SIB und referenziert einen Transformationsgraphen, der ein Eingabedokument transformieren soll. Der Untergraph ist somit entweder ein XSL- oder ein TT-Graph. Beides sind keine ausführbaren Graphen. Das normale Verhalten eines Standard-Graph-SIBs, das einen ausführbaren SIB-Graphen referenziert, wäre die Ausführung des Untergraphen. Führt eine Ausführungsumgebung den `Transform`-SIB aus, wird hingegen die `transform()`-Methode eines Transformer-Objekts aufgerufen. Nachfolgend wird näher erläutert, wie die `transform()`-Methode den Eingabebaum mit dem XSL- oder TT-Graphen transformiert.

In einem ersten Schritt übersetzt der Transformer den Graphen mit der `modelToXML`-Methode in ein Stylesheet. Sie wird auch für das `TTModelToXML`-SIB verwendet. jABC hält SIB-Graphen als Objekte vom Typ `SIBGraphModel` im Speicher. Ein spezieller SAX-Parser (`TTGraphSAXParser`) übersetzt ein `SIBGraphModel`-Objekt in SAX-Events, die eine XML-Serialisierung des Graphen darstellen. Graph-SIBs, die auf ein XSL-Graph- oder TT-Graph-Fragment zeigen, werden bei diesem Prozess über einen Expansionsvorgang in den Hauptgraphen integriert. Sie gehören zu dem resultierenden Dokument. Die speziellen Graph-SIBs `Template` und `Function` referenzieren hingegen Untergraphen, die Stylesheetmodule darstellen. Instanzen dieser SIB-Klassen erhalten eine gesonderte Behandlung. Der Untergraph wird nicht expandiert, sondern es werden der eindeutige Identifikator und der Typ des Subgraphen («`xmlgraph`» oder «`ttgraph`») als Attribute an das XML-Element angefügt, das den SIB repräsentiert. Auf diese Weise können sie bei dem Generierungsprozess des Transformationsgraphen in eine `<xsl:import>`-Deklaration übersetzt werden, die den entsprechenden TT- oder XSL-Graphen referenziert.

Die Implementierungen des `Transformer`-Interfaces haben interne Stylesheets für die Transformation des serialisierten XML-Dokuments in ein Stylesheet. Der `BS1Transformer` kann nur XSL-Graphen in Stylesheets übersetzen und verwendet hierzu die Starttransformation. Der `BS2Transformer` überschreibt die Methode, die im `BS1Transformer` die Umwandlung von Transformationsgraphen nach XSLT durchführt. Sie verwendet ebenfalls die Starttransformation für die Übersetzung von XSL-Graphen. Ferner verwendet sie TT-Compiler-Level-1, um TT-Graphen zu übersetzen. Der `DefaultTransformer` verwendet wiederum für die Umwandlung beider Graph-Typen die Stylesheets TT-Compiler-Level-2 und XSL-Compiler-Level-2, die in Stufe zwei des Bootstrappings erzeugt wurden.

Ein Transformationsgraph (XSL- oder TT-Graph) kann von einem anderen Graphen importiert werden. Dies geschieht in TT-Graphen implizit über die Verwendung von `Function`- und `Template`-SIBs und in XSL-Graphen in `Import`-SIBs. Diese Referenzen müssen in dem übersetzten Stylesheet realisiert und aufgelöst werden. Die Graphen der ersten beiden Bootstrapping Schritte verwei-

sen zum Beispiel auf abhängige Transformationsgraphen beziehungsweise Stylesheetmodule, da der Transformationsprozess von TT-Graphen in XSLT umfangreich ist. Für den Import von Stylesheets, repräsentiert durch Graphen, bietet Tree-Transformer eigene URI-Formate. XML, XSL- und TT-Graphen werden durch URIs der Form «`xmlgraph://$modelid`» oder «`ttgraph://$modelid`» referenziert. Die Transformer verwenden für die Umwandlung der Referenzen in Stylesheets jeweils eine Implementierung des `URIResolver`-Interfaces. Sie bedienen sich der `modelToXML()`-Methode des Transformers. Der URI-Resolver der `BS1Factory` ist einer einheitlichen Namensgebung folgend der `BS1URIResolver`.

In Stufe zwei werden die Stylesheets für den `DefaultTransformer` generiert. Im Unterschied zu den ersten beiden Schritten wird nicht nur das Hauptstylesheet generiert, sondern auch alle abhängigen Module. Sie liegen zur Laufzeit auf dem Klassenpfad und werden für die Umwandlung von Transformationsgraphen im «`default`»-Modus verwendet. Die abhängigen Stylesheets auf dem Klassenpfad werden über URIs der Form «`classpath://$path`» referenziert. Der Pfad wird aus dem SIB-Parameter `href` des Start-SIBs gelesen.

Anschließend an die Generierung des Stylesheets, wird es auf das Eingabedokument angewendet. Die Ausgabe der Transformation wird im SIB-Adapter in den gemeinsamen Speicher gelegt.

Die `PureSIBFactory` befindet sich nicht im Framework-Modul, sondern im Pure-Modul. Die Factory bietet die Implementierung für die Verwendung außerhalb von jABC. Sie wird von Anwendungen verwendet, die mit Java-Class-Generator erstellt wurden. Der generierte SIB-Graph muss ein `Transform`-SIB enthalten. Die Anwendung hängt lediglich von dem Pure- und API-Modul, dem SIB-Adapter, der *Java-Runtime-Umgebung* (JRE) und dem Saxon XSLT-Prozessor ab. Es besteht weder eine Abhängigkeit zu dem Framework-Modul noch zu dem jABC-Framework. Neben der Generierung der Anwendung mit Genesys Java-Class-Generator, werden auch der Transformationsgraph und all seine abhängigen Stylesheet-Module generiert. Die Umwandlung des Graphen in ein Stylesheet wird somit vorher ausgeführt und nicht, wie bei der Verwendung der Ausführungsumgebung von jABC, zur Laufzeit. Dieser Ansatz ist weniger flexibel, da Änderungen am Transformationsgraphen eine erneute Generierung erfordern. Im Gegenzug fällt ein Transformationsschritt weg und es wird kein Code zur Umwandlung von Graphen nach XSLT benötigt. Generierte Transformationsgraphen sind daher schneller und benötigen weniger Speicher, sowohl auf dem Speichermedium als auch im flüchtigen Speicher.

Das Hauptstylesheet und seine Module werden von dem Klassenpfad geladen. Die Pure-SIB-Factory bietet einen URI-Resolver, der URIs mit dem Aufbau «`classpath://$path`» vom Klassenpfad lädt. Der Pfad entspricht dem Wert des SIB-Parameters `href` von dem Start-SIB eines Transformationsgraphen. Da das `Transform`-SIB ein Graph-SIB ist, erhält die `transform`-Methode des Transformers den eindeutigen Bezeichner (UID) des referenzierten Transformations-

graphen. Das Hauptstylesheet liegt im Klassenpfad unter «`/$modelid.xsl`» und kann somit mit der Model-ID, als einzige Information, gefunden werden.

Das `ModelToXML`-SIB ruft über den SIB-Adapter die Methode `modelToXML()` des Transformers auf. Der Eingabegraph wird über eine URI festgelegt, die einen Transformationsgraphen referenziert. Das resultierende XSLT-Stylesheet wird als Zeichenkette in den gemeinsamen Speicher gelegt.

Die `loadTTModel`-Methode eines Transformers wird von dem `LoadTTModel`-SIB verwendet. Die Methode ruft für den über eine URI referenzierten Transformationsgraphen den `TTGraphSAXParser` auf. Die SAX-Events werden in eine Zeichenkette serialisiert und im gemeinsamen Speicher abgelegt. Das XML-Dokument, das erzeugt wird, entspricht einer Serialisierung eines `SIBGraphModel`-Objekts. Das ist nicht zu verwechseln mit dem Dokument, das der `ModelToXML`-SIB zurückgibt. Letzterer generiert die Repräsentation des Transformationsgraphen in XSLT.

4.6 Testmethoden

Für die Realisierung des Tree-Transformer-Plugins werden verschiedene Testmethoden eingesetzt. Diese werden in den Erstellungsprozess integriert, damit bei jedem Release sichergestellt ist, dass die Komponenten von Tree-Transformer funktionsfähig sind.

Die Implementierung in Java wird mit *Unit-Tests* geprüft. Sie rufen die Methoden mit verschiedenen Werten auf, so dass möglichst alle Kontrollstrukturen aufgerufen und kritische Randfälle überprüft werden.

Die Transformationsgraphen des Bootstrappings lassen sich nicht auf diese Weise testen, da es die entsprechenden Werkzeuge für XSLT und jABC nicht gibt. Tree-Transformer verwendet daher ein eigenes Verfahren, um sicherzustellen, dass die Stylesheets TT- und XSL-Graphen nach XSLT übersetzen können.

Nach dem Bootstrapping wird der ausführbare Graph, mit dem die zweite Stufe des Bootstrappings ausgeführt wurde, mit einer anderen Parameterisierung erneut aufgerufen. Dieses Mal wird die «`default`»-Implementierung gewählt, damit die zuvor erzeugten finalen Stylesheets, XSL-Compiler-Level-2 und TT-Compiler-Level-2, verwendet werden. Das Ergebnis der erneuten Generierung kann mit den XSLT-Dokumenten, die in der zweiten Stufe erzeugt wurden, verglichen werden. Sind die Ergebnisse gleich, hat das Bootstrapping Stylesheets erzeugt, die sich selbst übersetzen können. Da die Stylesheets aus Transformationsgraphen erzeugt werden, werden gleichzeitig viele Funktionen der Graphtypen getestet.

Abbildung 4.15 zeigt das erweiterte Bootstrapping. Die ersten drei SIBs sind unverändert[2]. Anschließend an das Bootstrapping werden die beiden Graphen

[2]Die SIB-Parameter für diese SIBs werden nicht angezeigt, da sie bereits in Abbildung 4.4

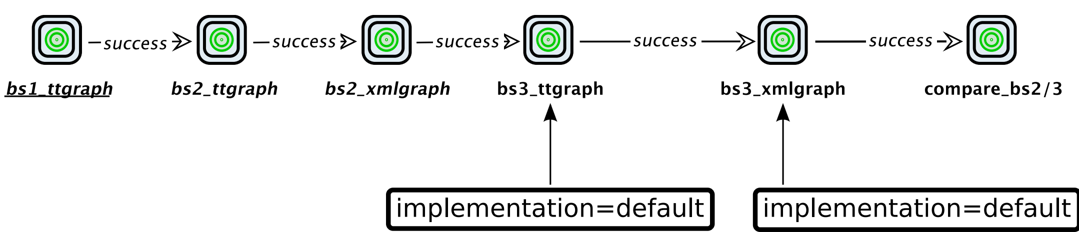

Abbildung 4.15: Testen des Bootstrappings

für die Transformation von TT-Graphen nach XSLT und für die Transformation von XSL-Graphen nach XSLT mit der Standardimplementierung erneut ausgeführt. Dieser Vorgang wird als Stufe 3 oder Teststufe bezeichnet. Der letzte SIB verweist auf einen Untergraphen, in dem der Vergleich der Stylesheets und ihrer Module mit denen aus der zweiten Stufe des Bootstrappings vorgenommen wird.

In mehreren Durchläufen des erweiterten Bootstrappings hat sich herausgestellt, dass die Laufzeit der Teststufe halb so hoch ist, wie bei der zweiten Stufe des Bootstrappings. Das hängt damit zusammen, dass bei Ersterem die Übersetzung importierter Transformationskomponenten wegfällt.

Mit dem erweiterten Bootstrapping lässt sich das Zusammenspiel der Funktionen testen. Zusätzlich werden einige Test-Transformations-Graphen auf einer vordefinierten Eingabe ausgeführt. Sie decken einzelne Funktionen der Graphen ab. Zum Beispiel könnte ein Test-Graph prüfen, ob XSL-Funktionsaufrufe in TT-Graphen mit keinem Parameter, einem Parameter und mehreren Parametern funktioniert. Die Ausgabedokumente werden zu Beginn manuell überprüft. Anschließend werden sie in folgenden Tests für den automatischen Vergleich verwendet, um die Funktionen auch über den Entwicklungsprozess des Plugins hinweg überwachen zu können. So können Fehler besser identifiziert werden, als in einem umfangreichen Test, wie dem Bootstrapping.

dargestellt werden.

5 Anwendungsbeispiel

Dieses Kapitel zeigt an einem Beispiel, wie Tree-Transformer eingesetzt werden kann. Es soll einfach verständlich sein und die Möglichkeiten des Konzepts zeigen. Dazu gehört unter anderem die Unterstützung der Aufteilung von Transformationen in Komponenten, die in verschiedenen Kontexten wiederverwendet werden können.

Tree-Transformer unterstützt einen XSLT-Experten bei der Erstellung von XSL-Graphen durch einen vereinfachten Entwurfsprozess und der Möglichkeit, Stylesheetfragmente zu kapseln. Dennoch ist diese Ebene der Modellierung als kanonische Abbildung von XSLT sehr technisch und konkret. Beispiele für XSL-Graphen werden daher schnell umfangreich und sind schwer zu beschreiben. Weiterhin ist die Syntax für einen XSLT-Experten intuitiv und ist durch die Erklärungen und das Beispiel in Kapitel 3.4 hinreichend beschrieben. Die LPC-Ebene ist für einen LPC-Nutzer ebenfalls intuitiv und wird in Kapitel 3.2 beschrieben. Der interessante und neue Ansatz ist vor allem die zweite Modellierungsebene. Daher ist dieses Kapitel auf die Sicht eines Transformationsdesigners konzentriert.

Die Starttransformation ist ein manuell erstelltes Stylesheet, das im Verlauf des Bootstrappings durch einen TT-Graphen ersetzt werden soll. Dies ist ein gutes Beispiel, um zu zeigen, wie ein bestehendes XSLT-Dokument in Tree-Transformer importiert, weiterverwendet und erweitert werden kann. Es ist dem Bootstrappingvorgang entnommen. Das Kapitel 4.2 gibt eine grundlegende Einsicht in das Thema.

Der TT-Graph soll zusätzlich zu den XML-SIBs, die die Knotentypen des XDM repräsentieren, noch den `Element`-SIB, der von der Importfunktion verwendet wird, nach XML übersetzen können. Bei der Realisierung des Plugins ist ein anderer Ansatz gewählt worden, als der hier beschriebene. Ein Grund hierfür ist, dass zu diesem Zeitpunkt die Importfunktion noch nicht zur Verfügung stand. Weiterhin wird auf Details der Implementierung verzichtet, da sie das Beispiel nur umfangreicher und schlechter verständlich machen würden.

In Kapitel 5.1 wird grob das Format beschrieben, in das Tree-Transformer Transformationsgraphen serialisiert. Kapitel 5.2 gibt einen kurzen Einblick in die Starttransformation. In Kapitel 5.3 wird die Modellierung der Transformation mit Tree-Transformer aus der Sicht eines Transformationsdesigners beschrieben. Abschließend geht Kapitel 5.4 darauf ein, wie Komponenten wiederverwendet und erweitert werden können.

5.1 XML-Serialisierung

Die Transformation von XSL-Graphen wird auf ihrer XML-Serialisierung durchgeführt. Jede SIB-Instanz wird durch ein XML-Element mit der Bezeichnung `<sib-instance>` repräsentiert, das den eindeutigen Bezeichner, die Parameter und die eingehenden und ausgehenden Kanten der Instanz sowie die UID des SIB-Typs enthält. Im Folgenden wird das Akronym „UID" für den eindeutigen Bezeichner eines SIB-Typs verwendet. SIB-Instanzen erhalten ebenfalls eine ID, die im weiteren Verlauf mit „Instanz-ID" bezeichnet wird. Die Zugehörigkeit einer SIB-Instanz zu einem SIB bestimmt die UID.

Die Kanten und Parameter werden als Kindelemente realisiert. Eine Kante hat immer genau ein Quell- und ein Ziel-SIB. Bei einer ausgehenden Kante wird das Ziel-SIB über seine Instanz-ID referenziert. Eingehende Kanten verweisen auf die Instanz-ID des Quell-SIBs. Die XML-Serialisierung kann auch allgemeine Graphen darstellen, weil die Graphstruktur über Referenzen aus Instanz-IDs realisiert wird.

XML-Fragmente, die durch Graph-SIBs gekapselt sind, wie zum Beispiel die XSL-SIBs, werden bei der Serialisierung expandiert. Das bedeutet, der Graph-SIB wird durch den Untergraphen ersetzt. Der `DocumentNode`-SIB des Untergraphen wird bei diesem Vorgang gelöscht und alle eingehenden Kanten das Graph-SIBs zeigen auf die SIB-Instanz, auf die die `child`-Kante des `DocumentNode`-SIBs gezeigt hat. Die Expansion aller Untergraphen führt dazu, dass ein serialisierter XSL-Graph nur aus XML-SIBs besteht, da alle Graph-SIBs ersetzt wurden.

5.2 Starttransformation

Das manuelle Stylesheet definiert für die XML-SIBs jeweils eine Template-Regel, die den zugehörigen Knoten in der Ausgabe erzeugt. Hierbei wird auf die Verarbeitung von Processing-Instructions und Kommentare verzichtet, da sie für das Bootstrapping nicht benötigt werden. Eine Instanz von dem `TextNode`-SIB wird zum Beispiel als Textknoten in das Ausgabedokument geschrieben. Der Wert des SIB-Parameters `text` ist der Inhalt des Knotens. Für einen `ElementNode`-SIB, werden neben der Erzeugung des Elementknotens noch die Attribute, Namensräume und Kinder des Elements weiterverarbeitet. Hierzu werden die SIBs, beziehungsweise die XML-Elemente, durch die sie repräsentiert werden, der entsprechenden Achse selektiert und Template-Regeln für sie aufgerufen. Die Selektion aller SIBs auf einer Achse wird über die XSL-Funktion `select-axis()` realisiert. Weitere XSL-Funktionen übernehmen Standardaufgaben, wie zum Beispiel das Erzeugen eines qualifizierten Namens der Form «`prefix:localname`» aus den SIB-Parametern `prefix`

und `localName` von den XML-SIBs `ElementNode`, `AttributeNode` und `ProcessingNode`.

5.3 Modellierung des TT-Graphen

Ein XSLT-Experte teilt die Template-Regeln und XSL-Funktionen in einzelne Stylesheetmodule auf. Diese können mit der Importfunktion in XSL-Graphen übersetzt werden. Der TT-Teil-Graph in Abbildung 5.1 zeigt, wie ein Transformationsdesigner die Template-Regeln zu einer Gruppe zusammenfügen kann. Da es sich um einen Teilgraphen handelt und nicht um einen vollständigen TT-

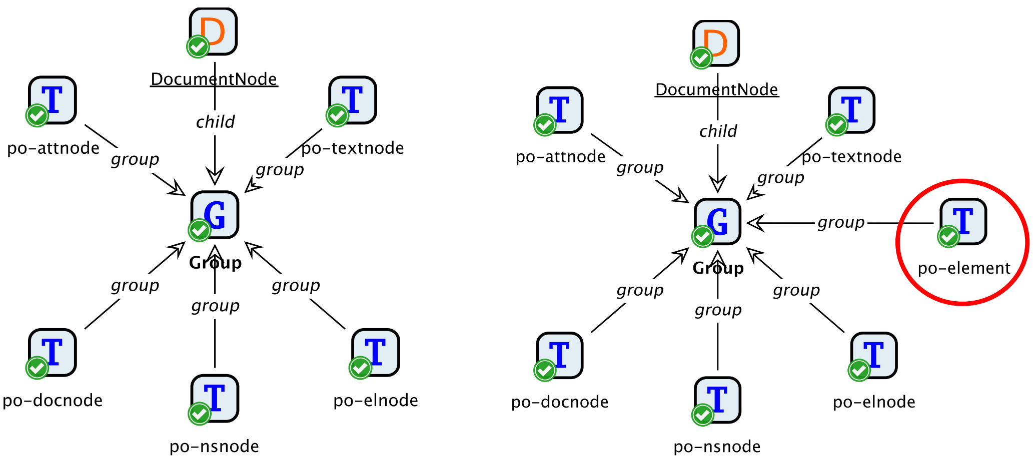

Abbildung 5.1: Gruppe von Template-Regeln für die Verarbeitung von XML-SIBs

Abbildung 5.2: Erweiterte Gruppe von Template-Regeln für die Verarbeitung von XML-SIBs

Graphen, enthält der Graph kein `TemplateMain`- oder `FunctionMain`-SIB. Damit das Plugin den Graphen dennoch erkennt, enthält er einen `DocumentNode`-SIB. Diese Konvention ist daraus abgeleitet, dass im XPath-2.0-Datenmodell Bäume einen Dokumentknoten als Wurzel haben. Die `child`-Kante zeigt auf den `Group`-SIB. So werden eingehende Ergebniskanten in einem Graphen, der eine Hierarchieebene höher ist, auf diesen SIB geleitet. Sie bestimmen die Eingabesequenz der Gruppe.

Die `Template`-SIBs zeigen mit der `group`-Achse auf das `Group`-SIB. Das definiert die Zugehörigkeit zu dieser Gruppe. Der `Group`-SIB sucht für jeden Knoten aus seiner Eingabesequenz eine Template-Regel aus der Gruppe und führt sie für den Knoten aus.

Die Namen der `Template`-SIBs entsprechen dem Namen des referenzierten Modells. Jeder verweist auf eines der importierten Stylesheetmodule. Sie verarbeiten ein XML-Element, das ein XML-SIB repräsentiert. Der Transformations-

graph mit dem Namen «`po-attnode`» verarbeitet zum Beispiel `AttributeNode`-SIBs.

Damit die Gruppe von Template-Regeln auch `Element`-SIBs verarbeiten kann, erteilt der Transformationsdesigner dem XSLT-Experten den Auftrag, die entsprechende Komponente zu erstellen. In Abbildung 5.2 ist der erweiterte TT-Teil-Graph dargestellt, der die zusätzliche Template-Regel in die Gruppe einbindet. An dieser Stelle könnten weitere Template-Regeln eingebunden werden, die Processing-Instructions und Kommentare unterstützen. Der Graph trägt die Bezeichnung «`sg-po-group`».

In Abbildung 5.3 ist der TT-Graph gezeigt, der die Starttransformation ersetzt und um die Funktion erweitert auch `Element`-SIBs verarbeiten zu können. Der Graph in Abbildung 5.2 wird über einen Graph-SIB eingebunden.

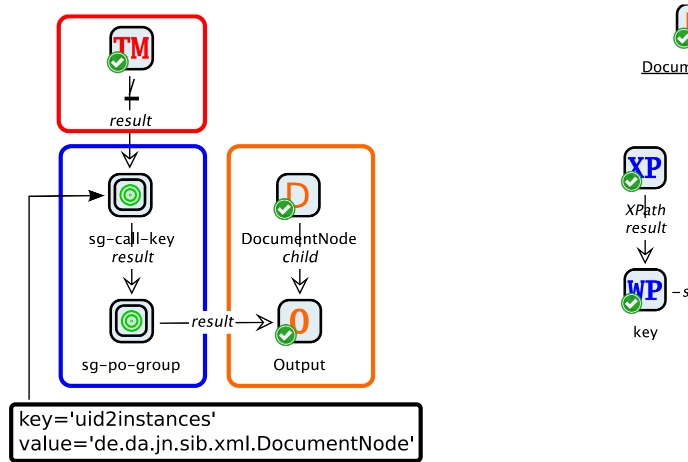

Abbildung 5.3: TT-Graph für die Transformation von XSL-Graphen nach XSLT

Abbildung 5.4: Untergraph des Graph-SIBs «`sg-call-key`» aus Abbildung 5.3

Der `TemplateMain`-SIB verarbeitet den Dokumentknoten des Eingabedokuments und gibt ihn an den Untergraphen mit der Bezeichnung «`sg-call-key`» weiter. Dieser Graph ist in Abbildung 5.4 zu sehen. Er ruft eine XSL-Funktion auf, die eine UID den XML-Elementen der entsprechenden SIB-Instanzen zuordnet und zurückgibt. Der Aufruf der XSL-Funktion wird von dem `Function`-SIB mit der Bezeichnung «`f-call-key`» dargestellt. Die referenzierte XSL-Funktion ist eine Komponente, die durch den XSLT-Experten erstellt wurde. Sie benötigt drei Parameter, die mit `WithParameter`-SIBs modelliert werden.

Der erste Parameter hat den Namen «`key`». Er bestimmt den Typ von Schlüssel-Wert-Paaren, auf den bei diesem Funktionsaufruf zugegriffen werden soll. Ein `XPath`-SIB erzeugt die Eingabe. Der Pfadausdruck wird über einen Modellparameter in dem Graph-SIB in Abbildung 5.3 gesetzt.

Der Anzeigename des SIBs ist «`XPath`», da der Pfadausdruck nicht in dem Untergraphen gesetzt ist. Der SIB-Parameter `key` des Graph-SIBs mit der Bezeichnung «`sg-call-key`» setzt den Wert auf «`'uid2instances'`». Damit wird festgelegt, dass die XSL-Funktion auf die Schlüssel-Wert-Paare zugreifen soll, die UIDs den zugehörigen SIB-Instanzen zuweisen.

Der zweite Parameter «`keyvalue`» setzt die UID, zu der die SIB-Instanzen gesucht werden sollen. Er wird ebenfalls über einen `XPath`-SIB gesetzt, dessen Pfadausdruck über einen Modellparameter mit der Bezeichnung «`value`» bestimmt wird. In Abbildung 5.3 ist zu sehen, das der SIB-Parameter den Wert «`'de.da.jn.sib.xml.DocumentNode'`» hat. Die Funktion liefert somit alle XML-Elemente, die ein `DocumentNode`-SIB repräsentieren, aus dem Eingabedokument, dass durch den Dokumentknoten definiert ist. Letzterer wird mit dem dritten Parameter übergeben.

Der dritte Parameter erhält seine Eingabe von einem `XPath`-SIB. Dieser führt den Pfadausdruck «`root($input)`» auf dem Dokumentknoten des Eingabedokuments aus, der über den `TemplateMain`-SIB des darüberliegenden Graphen übergeben wurde. Die XSL-Funktion `root()` gehört zu den integrierten Funktionen von XPath und liefert den Dokumentknoten des übergebenen Knotens. In diesem Fall liefert er den Knoten selbst zurück. Die `result`-Achse des `Function`-SIBs wird als Modelbranch exportiert.

Das XML-Element `<sib-instance>`, das den `DocumentNode`-SIB des Eingabedokuments repräsentiert, wird von der XSL-Funktion zurückgegeben und über den aufrufenden Graph-SIB an den nächsten Graph-SIB mit der Bezeichnung «`sg-po-group`» weitergereicht. Dieser SIB verweist auf den Graphen in Abbildung 5.2 und übersetzt den gesamten XSL-Graphen ausgehend von dem `DocumentNode`-SIB nach XSLT. Der `Group`-SIB veröffentlicht die `result`-Achse. Sie wird in dem aufrufenden Graphen an einen `Output`-SIB weitergegeben.

Der TT-Graph in Abbildung 5.3 hat nur eine mögliche Ausgabe und enthält daher keine `If`-SIBs. Die Ausgabe ist das XSLT-Dokument, das den XSL-Graphen repräsentiert, der als Eingabedokument verwendet wurde.

5.4 Wiederverwendung von Komponenten

Alle Schritte des Bootstrappings transformieren die XML-Serialisierung von SIB-Graphen. Das bedeutet, dass die Eingabedokumente das gleiche Format haben und sich die Transformationen in der gleichen Problemdomäne befinden. In dieser lassen sich verschiedene Aufgaben identifizieren, die an unterschiedlichen Stellen benötigt werden. Tree-Transformer ermöglicht, sie in Komponenten zu isolieren und an mehreren Stellen ohne zusätzlichen Aufwand einzusetzen. Ferner ermöglicht der Ansatz Aufgaben, die ähnlich sind, auf ihre Schnittmenge zu reduzieren, die wiederverwendet werden kann.

Die Übersetzung von XML-Graphen nach XML wird sowohl für XSL-Graphen als auch für TT-Graphen benötigt. Erstere sind XML-Graphen, da XSLT in XML notiert wird. Letztere deklarieren mit XML-SIBs die statische Ausgabe (Ergebniselemente) in den Deklarationen der möglichen Ausgaben. Zusätzlich werden dynamische Ausgaben über `Output`-SIBs realisiert, die ihre Eingabesequenz in die Ausgabe kopieren. Die Übersetzung von XML-Graphen nach XML kann demnach für TT-Graphen wiederverwendet werden, wobei zusätzlich `Output`-SIBs verarbeitet werden müssen.

Der Teilgraph aus Abbildung 5.2 wird zum Beispiel in dem Graphen in Abbildung 5.5, für die Übersetzung von TT-Graphen nach XSLT, über einen Graph-SIB referenziert. Die Komponente zur Verarbeitung des zusätzlichen SIBs vom

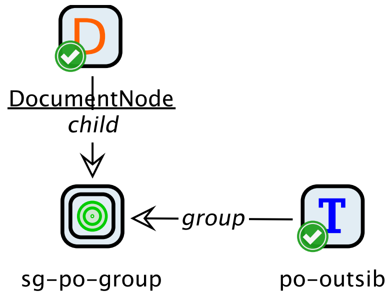

Abbildung 5.5: Testen des Bootstrappings

Typ `Output` kann durch einen XSLT-Experten implementiert werden. Es muss lediglich die Eingabesequenz, die durch die eingehenden Ergebniskanten definiert ist, in die Ausgabe kopiert werden. Die neue Template-Regel wird über die `group`-Achse der Gruppe des Untergraphen hinzugefügt. Der erweiterte Teilgraph kann für die Übersetzung von TT-Graphen nach XSLT verwendet werden. Der Graph mit der Bezeichnung «`sg-po-group`» kann unverändert in den Graphen aus Abbildung 5.3 eingesetzt werden.

6 Ergebnisse und Ausblick

Dieses Kapitel fasst die Ergebnisse der Diplomarbeit zusammen, geht auf die Probleme bei der Entwicklung des Konzepts und der Implementierung von Tree-Transformer ein und gibt einen Ausblick auf die Möglichkeiten, die sich durch diese Arbeit in Zukunft ergeben.

6.1 Ergebnisse

Mit Tree-Transformer können Transformationsgraphen modelliert und in ausführbare SIB-Graphen eingebunden werden. Der Entwurfsprozess kann in drei Modellierungsebenen unterteilt werden:

LPC-Ebene
Die Transformationsgraphen können ohne technische Kenntnisse in ausführbare SIB-Graphen eingebunden werden.

TT-Ebene
Mit TT-Graphen lassen sich Transformationen abstrakt graphisch beschreiben, indem Transformationskomponenten koordiniert werden. Es sind keine Detailkenntnisse der Zielsprache XSLT notwendig.

XSL-Ebene
Mit XSL-Graphen können die Transformationskomponenten für TT-Graphen technisch realisiert werden. Sie sind eine graphische Darstellung von XSLT. Es wird kein Überblick über die Transformation oder die Anwendung benötigt.

Die klare Trennung der Modellierungsebenen ermöglicht eine Rollenverteilung in den Anwendungsexperten (LPC-Ebene), den Transformationsdesigner (TT-Ebene) und den XSLT-Experten (XSL-Ebene). Durch die Kapselung von Stylesheetmodulen in Komponenten der TT-Ebene steht der vollständige Funktionsumfang von XSLT zur Verfügung. Weiterhin ist die Darstellung der TT-Graphen eine Abstraktion der Zielsprache.

Der Entwurfsprozess von Transformationen wird durch den Einsatz von LocalChecker und GEAR in allen drei Ebenen vereinfacht. Die Prüfmethoden des LocalChecker-Plugins testen lokale Bedingungen auf den SIBs und ihren direkten Vorgängern und Nachfolgern. Mit GEAR werden globale Bedingungen an

einem konkreten Transformationsgraphen überprüft. Die Tests werden während der Modellierung eines SIB-Graphen durchgeführt und informieren den Anwender über Fehler. Ferner ist die Modellierumgebung restriktiv entworfen worden, so dass viele Syntaxfehler, die bei der Editierung in einem XSLT-Editor auftreten, nicht entstehen. Diese Funktionen helfen, Fehler frühzeitig zu erkennen und zu beheben. So kann in vielen Fällen darauf verzichtet werden, den XSLT-Prozessor aufzurufen, um Fehler zu finden.

Tree-Transformer bietet durch die Einbindung von Transformationsgraphen in ausführbare SIB-Graphen die Möglichkeit, mit Tracer eine Transformation schnell und einfach in jABC zu testen. Die Verwaltung der Ein- und Ausgabedokumente kann über Common-SIBs erfolgen. Weiterhin können auch LPC-Graphen, die Transformationen ausführen, mit Genesys in eine eigenständige Anwendung übersetzt werden. Der Aufruf des XSLT-Prozessors über eine XML-Processing-API, wie zum Beispiel JAXP, muss daher nicht mehr implementiert werden. Die Generatoren von Genesys sind nicht auf Java-Anwendungen beschränkt. Es können Anwendungen für andere Technologien oder Plattformen, wie zum Beispiel *Java-Servlets* für Webanwendungen oder *JME* für eingebettete Systeme, generiert werden.

Mit der Importfunktion können bestehende XSLT-Dokumente in XSL-Graphen umgewandelt werden. Dies ermöglicht zum Einen den einfachen Umstieg auf Tree-Transformer, falls in einem Projekt bereits Stylesheets vorhanden sind, die übernommen werden sollen. Zum Anderen gibt es einem XSLT-Experten die Freiheit, weiterhin seinen präferierten Editor für die Erstellung von Transformationskomponenten zu verwenden.

Die Implementierung in Java von Tree-Transformer ist sehr schlank, da viele Funktionen in ausführbaren SIB-Graphen und Transformationsgraphen realisiert sind. Die Umsetzung der Transformationen von XSL- und TT-Graphen nach XSLT mit einem Bootstrappingvorgang, hat gezeigt, dass auch umfangreiche Transformationen mit diesem Ansatz möglich sind. Das Bootstrapping (siehe Kapitel 4.2 auf Seite 76) und das Anwendungsbeispiel (siehe Kapitel 5 auf Seite 99) haben gezeigt, dass in dieser Problemdomäne mit Tree-Transformer viele Komponenten identifiziert, isoliert, erweitert und an mehreren Stellen wiederverwendet werden konnten. Die Funktionen des Plugins benötigen zum Beispiel größtenteils Transformationen auf der XML-Serialisierung von Transformationsgraphen. Hierfür lassen sich mehrere wiederverwendbare Transformationskomponenten identifizieren, wie die Selektion von SIB-Instanzen auf einer Achse. Diese können in XSL-Graphen gekapselt und bei der Erstellung von TT-Compiler-Level-1 und 2 sowie XSL-Compiler-Level-2 wiederverwendet werden.

Die Laufzeiten des Bootstrappings zeigen, dass die Compiler von Tree-Transformer effiziente Stylesheets erzeugen. Weiterhin ist der Quelltext von XSL-Compiler-Level-2, der aus einem TT-Graphen erstellt wurde, der Starttransformation sehr ähnlich. Beide haben die gleiche Semantik. Der

Overhead durch die Verwendung der TT-Graphen, die von XSLT abstrahieren, ist demnach gering.

6.2 Probleme

Einige Bereiche der Syntaxdefinitionen von TT- und XSL-Graphen sowie weitere Entscheidungen bei der Entwicklung des Tree-Transformer-Plugins, beinhalten Zugeständnisse an die verwendeten Technologien XSLT und jABC:

- Da eine Achse oder ein Branch in jABC nicht mehreren Kanten zugewiesen werden kann, ist der `Branch`-SIB mit in die TT-SIBs aufgenommen worden. Er verteilt seine Eingabesequenz über unbeschriftete ausgehende Kanten an die nachfolgenden SIBs, um die mehrfache Vergabe einer Kantenbeschriftung zu simulieren.

- jABC unterstützt keine getypten SIB-Graphen. Ansonsten könnte ein neu erstellter XSL-Graph bereits den obligatorischen `DocumentNode`-SIB enthalten. TT-Graphen könnten zu Beginn einen `TemplateMain`- oder `FunctionMain`-SIB beinhalten.

- In getypten Graphen könnte verboten werden, Modelbranches zu verschmelzen. Das ist in der aktuellen Implementierung immer erlaubt und führt bei XSL-Graphen zu XML-Dokumenten, die nicht wohlgeformt sind. Wenn zum Beispiel zwei `child`-Achsen eines Untergraphen verschmolzen werden und über einen Graph-SIB an Kindknoten gebunden werden, haben diese mehrere Elternknoten.

- In XSLT identifizieren Funktionsaufrufe ihre Parameter über die Reihenfolge, bei Template-Regeln geschieht dies über den Namen. Derzeit muss das in TT-Graphen vom Modellierer berücksichtigt werden. In einer späteren Implementierung kann die Reihenfolge von Parametern bei der Übersetzung nach XSLT aus den Namen erzeugt werden, da die referenzierte Deklaration der XSL-Funktion in einem Transformationsgraphen zur Verfügung steht.

Einige der Probleme mit jABC resultieren daraus, dass das ursprüngliche Modellierungskonzept für Kontrollflussgraphen entworfen wurde. Die mehrfache Verwendung einer Kantenbeschriftung, die in diesem Kontext eine Verzweigung repräsentiert, würde Nichtdeterminismus zulassen. Ab Version 4 von jABC sollen diese Probleme in Angriff genommen werden. Dabei sollen unter anderem getypte Graphen unterstützt werden, damit mehrere Konzepte, wie zum Beispiel LPC und LDC, besser koexistieren können.

6.3 Ausblick

Die Effizienzsteigerung, die sich aus der Rollenverteilung in Anwendungsexperte, Transformationsdesigner und XSLT-Experte ergeben soll, konnte im Zuge der Diplomarbeit nicht getestet werden, da der Autor alle drei Rollen übernommen hat. Zukünftige Projekte, die das Tree-Transformer-Plugin einsetzen, könnten hierfür weitere Daten liefern.

Der LocalChecker und GEAR können nicht alle Fehlerquellen bei dem Entwurf von Transformationsgraphen abdecken, weder syntaktisch noch semantisch. Letzteres würde ein Type-Checking [10] erfordern und ist nur für sehr eingeschränkte Transformationen und Datenschemata effizient. Für die syntaktische Überprüfung muss der XSLT-Prozessor verwendet werden. Der aktuell verwendete XSLT-Prozessor bietet ein Listener-Konzept, mit dem die Ausführung eines Stylesheets verfolgt und beeinflusst werden kann. Weiterhin kann der XPath-Kontext beobachtet werden. Aus dieser Funktion kann ein Debugger für Transformationsgraphen erstellt werden, der in jedem Schritt die Ausführung anhält und diesen im Transformationsgraphen visualisiert. Über den Debugging-Dialog von Tracer könnten die Schritte gesteuert werden.

Für die Fehlerbehandlung bietet der verwendete XSLT-Prozessor ebenfalls ein Listener-Konzept. Es könnte dazu verwendet werden, Kompilierfehler in Graphen zu visualisieren. Weiterhin könnte bei Laufzeitfehlern der Stack-Trace, dass heißt die zuletzt ausgeführten Instruktionen, bevor ein Fehler aufgetreten ist, im Graphen angezeigt werden.

Eine interessante Erweiterung von Tree-Transformer wäre ferner eine Unterstützung von XML Schema in Transformationsgraphen. Seit der Version XSLT 2.0 wird diese Sprache unterstützt, um Knoten über ihren Typen zu identifizieren sowie die Eingabe und Ausgabe gegen ein Schema zu prüfen. Zu dem Zeitpunkt der Erstellung der Arbeit existierte kein frei verfügbarer XSLT-Prozessor für Java, der Schemavalidierung unterstützt.

Weiterhin wäre es sinnvoll, einen Transformationsgraphen automatisch vorzukompilieren, wenn er gespeichert wird. Einerseits könnte der Anwender darüber informiert werden, wenn er eine fehlerhafte Transformation abspeichert, andererseits würde bei der Ausführung die Kompilierung wegfallen.

Abbildungsverzeichnis

2.1 Architektur des LPC-Konzepts 9
2.2 Das SIB-Adapter-Entwurfsmuster 13
2.3 DEA – Ausführung eines SIB-Graphen 16
2.4 Ausführung eines konkreten SIB-Graphen im Tracer 17
2.5 Abhängigkeiten zwischen den Strukturen im XPath Datenmodell 28

3.1 Modellierung einer Transformation in LPC-Graphen 43
3.2 Modellierung einer Transformation in einem Transformationsgraphen mit vordefinierten Funktions-SIBs für einzelne Aufgaben . 44
3.3 Ebenen der Modellierung 47
3.4 Beispiel eines ausführbaren Graphen 49
3.5 Beispiel eines TT-Graphen 52
3.6 Beispiel eines TT-Graphen 53
3.7 SIB-Icons der TT-SIBs 55
3.8 Beispiel eines XSL-Graphen 65
3.9 SIB Graph des XSL Template-Elements 66
3.10 SIB-Icons der XML-SIBs 67
3.11 Vereinfachter Entwurfsprozess 73

4.1 Bootstrapping-Stufe-1 78
4.2 Bootstrapping-Stufe-2 79
4.3 Generierung abhängiger Transformationsgraphen 81
4.4 Aufruf der Schritte des Bootstrappings 82
4.5 Laden des TT-Graphen für TT-Compiler-Level-1 82
4.6 Erstellung des TT-Compiler-Level-1 83
4.7 Erstellung des TT-Compiler-Level-2 83
4.8 Umwandlung eines XSL-Graphen in ein Stylesheet (Level 1 und 2) 85
4.9 Umwandlung eines TT-Graphen in ein Stylesheet (Level 2) . . . 85
4.10 UML-Klassendiagramm der Factory-Klassen 89
4.11 UML-Klassendiagramm der `TTPluginFactory` und ihrer Implementierung 90
4.12 UML-Klassendiagramm, das die lose Kopplung durch das SIB-Adapter-Entwurfsmuster zwischen ausführbaren SIBs und ihren Komponenten zeigt 91

4.13 UML-Klassendiagramm der `TTSIBFactory`, ihren konkreten Klassen und des `Transformer`-Interfaces 93
4.14 UML-Klassendiagramm der `Transformer`- und `URIResolver`-Realisierungen 93
4.15 Testen des Bootstrappings 97

5.1 Gruppe von Template-Regeln für die Verarbeitung von XML-SIBs 101
5.2 Erweiterte Gruppe von Template-Regeln für die Verarbeitung von XML-SIBs 101
5.3 TT-Graph für die Transformation von XSL-Graphen nach XSLT 102
5.4 Untergraph des Graph-SIBs «`sg-call-key`»aus Abbildung 5.3 . 102
5.5 Testen des Bootstrappings 104

Listings

2.1 Aufruf eines SIB-Adapters 14
2.2 XML-Deklaration 22
2.3 Wohlgeformtes XML-Dokument 22
2.4 Ausbalanciertes Dokument 23
2.5 Nicht ausbalanciertes Dokument 23
2.6 Namensraum-Deklarationen 24
2.7 Pfadausdruck mit zwei Schritten 29
2.8 expanded-QName() 31
2.9 Pfadausdruck 33
2.10 Pfadausdruck 33
2.11 Beispiel-Stylesheet 38
2.12 Eingabedokument 39
2.13 Ausgabedokument 40
3.1 Vereinfachte Variante des Stylesheets aus Listing 2.11 auf Seite 38 51
3.2 Ausgabedokument 52
3.3 Beispiel einer ungenutzten Variablen 61

Akronyme

Application Programming Interface (API)

Schnittstelle einer Anwendung oder Bibliothek nach außen

Business Process Execution Language (BPEL)

Ausführbare Modellierungssprache für Geschäftsprozesse

Document Object Model (DOM)

Programmierschnittstelle für den Zugriff auf XML-Dokumente, repräsentiert ein Dokument als Baum im Speicher

Document Type Definition (DTD)

Sprache zur Definition der Struktur eines XML-Dokuments

Game-based, Easy And Reverse model-checking (GEAR)

JavaABC Plugin: Ermöglicht Modelchecking auf SIB-Graphen

Java Application Building Center (jABC)

Graphisches Modellierungstool

Java API for XML Processing (JAXP)

Programmierschnittstelle für die Verarbeitung von XML

Java Micro Edition (JME)

Distribution von Java für eingebettete Systeme

Java Native Interface (JNI)

Programmierschnittstelle in Java für den Aufruf plattformspezifischer Methoden

Java Runtime Environment (JRE)

Laufzeitumgebung für Java

Lightweight Declarative Coordination (LDC)

Konzept der graphischen Koordinierung beziehungsweise Kombination von Deklarationen. Hierbei handelt es sich um eine Erweiterung des LPC-Konzepts auf deklarative Sprachen.

Lightweight Process Coordination (LPC)

Serviceorientiertes Konzept der Prozessmodellierung beziehungsweise -koordinierung von jABC für ausführbare SIB-Graphen. Es fügt eine Koordinationsschicht in das Drei-Schichten-Modell zwischen der Komponenten- und Präsentationsschicht ein.

Remote Method Invocation (RMI)

Protokoll in Java für den Aufruf entfernter Methoden

Scalable Vector Graphics (SVG)

Beschreibungssprache für Vektorgraphiken in XML-Syntax

Uniform Resource Identifier (URI)

Einheitlicher Identifikator für Ressourcen

Web Service Description Language (WSDL)

Beschreibungssprache für Web Services (Ein- und Ausgabeverhalten)

XPath Data Model (XDM)

Das Datenmodell von XPath

Extensible HyperText Markup Language (XHTML)

Hat die gleiche Ausdruckskraft wie HTML und ist XML-konform

Extensible Markup Language (XML)

Erweiterbare Markup-Sprache für dokumentzentrierte, semistrukturierte und strukturierte Daten

XML Path Language (XPath)

Expression Language, die in verschiedene Sprachen der XML-Familie, wie zum Beispiel XQuery, XPointer, XSLT und DOM eingebettet ist. Sie dient unter anderem der Adressierung von Daten in einem XML-Dokument.

Extensible Stylesheet Language Formatting Objects (XSL-FO)

Transformationssprache für XML. Erweitert XML Dokumente um Formatierungshinweise. Wird beispielsweise verwendet, um XML- in PDF-Dokumente zu übersetzen.

Extensible Stylesheet Language Transformations (XSLT)

XML-Skriptsprache zur Transformation von XML-Bäumen

Index

Abstract Factory, 88
API, 18

Corba, 13

DEA, 16
Default-Namespace, 24
Document-Type-Definition, 23
DOM, 25
DTD, 23

Encoding, 21

HTML, 40

jABC, 7
  Anwendungsexperte, 7
  Ausführbarer SIB-Graph, 8
  Branches, 9
  Common-SIBs, 12, 42, 49
  erweiterbare Branches, 11
  Framework, 8
  Genesys
    Service-Adapter, 19
  GraphSIB
    Programmieren, 15
  LPC, 9
  Modellargumente, 15
  mutable Branches, 11
  Plugin
    GEAR, 20
    Genesys, 19
    LocalChecker, 20
    SIBCreator, 21
    Tracer, 19
  Projekt, 8
  Service Independent Building Block, 8
  SIB, 8, 11
    UID, 67
  SIB-Adapter, 12
  SIB-Argument, 11
  SIB-Experte, 8
  SIB-Graph, 7
  SIB-Instanz, 11
  SIB-Pattern, 12
  Start-SIB, 16
Java
  Annotation, 7, 11
  Classpath, 15
  Ereignis, 18
  Event, 18
  Interface, 12
  Listener, 18
  Package, 67, 70
  Properties, 15
  Thread-Pool, 18
Java APIs for XML Processing, 4
Java Collections Framework, 81
Java List, 81
Java-Class-Extruder, 19
Java-Micro-Edition, 47
Java-Property, 89
Java-Pure-Generator, 19
Java-Reflection-API, 75
Java-Runtime-Umgebung, 95
Java-Servlet, 106
JavaABC

Modell-Listener, 20
SIB
Instanz, 70
JAXP, 4
JNI, 13
JRE, 95

Lightweight-Declarative-Coordination, 46
Logger, 20

Namensraum, 24
Namespace, 24
Nebenläufigkeit
Race-Condition, 17
Thread, 16

Programmierung
by reference, 28
by value, 28
Liste, 27

RMI, 13

Standard-Namensraum, 24
StAX-Parser, 88

Template-Regel, 23
Type Checking, 31
Typtest, 31

Unit-Test, 96
URI, 24

Variable
Scope, 34
Sichtbarkeit, 34

XML
anyURI, 32
Document-Object-Model, 25
Dokumentelement, 22
Dokumentreihenfolge, 71
DOM, 25, 43
QName, 31
Wohlgeformtheit, 68
XML Schema
Restriction, 32
Restriktion, 32
Simple Type, 31
XML-Schema
Complex Type, 31
Einfache Typen, 31
Komplexe Typen, 31
XML-Schema-Definition, 23
XPath
Achse, 29, 58
Axis, 29, 58
Document Order, 32
Dokumentreihenfolge, 32
Evaluation-Context, 27
Knotentext, 29
Node Test, 29
Schritt, 29
Sequence, 27
Sequence-Expressions, 28
Sequence-Type-Constructor, 32
Sequenz, 27
Sequenzausdruck, 28
Steps, 29
Typ-Test-Ausdruck, 32
Typdeskriptoren, 32
Type-Expressions, 32
XPath 2.0, 25
XSLT, 34
algorithmisches Entwurfsmuster, 35
Befehl, 70
Benanntes Template, 37
Built-In Template-Rule, 36
Computational Design-Pattern, 35
Default-Modus, 36
Ergebniselement, 36
Function, 37
Funktion, 37
Instruction, 34, 70

Instruktion, 34
Integrierte Template-Regel, 36
Literal-Result-Element, 36
Modus, 36
Muster, 25
Named Template, 37
Navigational Design-Pattern, 35
Pattern, 25
Regelbasiertes Entwurfsmuster, 35
Rule-Based Design-Pattern, 35
Sequence Constructor, 35
Sequenzkonstruktor, 35
Stylesheet-Modul, 38
Template-Regel, 36
  Priorität, 36
Template-Rule, 36
  Priority, 36
Top-Level-Deklaration, 36, 70
XSLT-Pattern, 56

Literaturverzeichnis

[1] Apt, Krzysztof R. und Ernst-Rüdiger Olderog: *Verification of sequential and concurrent Programs*. Springer-Verlag, 1991.

[2] Calic, Michael, Börje Sieling und Petra Simon: *Grundbegriffe der Objektorientierung*. HMD - Praxis Wirtschaftsinform., 210:7–22, 1999.

[3] Foundation, The Apache Software: *Apache Maven Project*. `http://maven.apache.org/`.

[4] Heesch, Dimitri van: *Doxygen. Source code documentation generator tool*. `http://www.doxygen.org`.

[5] Herbert Schildt: *C++ Die professionelle Referenz*. Mitp-Verlag, September 2004.

[6] Jipping, Michael J. und Kim B. Bruce: *The Imperative Language Paradigm*. In: *The Computer Science and Engineering Handbook*, Seiten 1983–2005. 1997.

[7] Jörges, Sven, Christian Kubczak, Ralf Nagel, Tiziana Margaria und Bernhard Steffen: *Model-Driven Development with the jABC*. In: *HVC - IBM Haifa Verification Conference*, 2006.

[8] Manthey, Rainer: *Declarative Languages - Paradigm of the Past or Challenge of the Future?* In: *East/West Database Workshop*, Seiten 1–16, 1990.

[9] Margaria, Tiziana und Bernhard Steffen: *Service Engineering: Linking Business and IT*. IEEE Computer, 39(10):45–55, 2006.

[10] Martens, Wim und Frank Neven: *Frontiers of Tractability for Typechecking Simple XML Transformations*. In: *PODS*, Seiten 23–34, 2004.

[11] McLaughlin, Brett D. und Justin Edelson: *Java and XML, Solutions to Real-World Problems (3rd Edition)*. O'reilly, 2006.

[12] Michael Kay: *XPath 2.0 Programmer's Reference*. Wiley Publiching, Inc, August 2004.

[13] Michael Kay: *XSLT 2.0 Programmer's Reference, Third Edition*. Wiley Publiching, Inc, August 2004.

[14] Müller-Olm, Markus, David A. Schmidt und Bernhard Steffen: *Model-Checking: A Tutorial Introduction*. In: *SAS*, Seiten 330–354, 1999.

[15] Nagel, Ralf et al.: *Java ABC Framework Homepage*. `http://www.jabc.de`.

[16] Reinhard Diestel: *Graphentheorie*. Springer, Berlin, März 2006.

[17] SUN: *Java^{TM} 2 Platform Standard Edition 5.0 API Specification*. `http://java.sun.com/j2se/1.5.0/docs/api/`.

[18] TC, OASIS Web Services Business Process Execution Language (WSBPEL): *Web Services Business Process Execution Language Version 2.0*. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`.

[19] Ullenboom, Christian: *Java ist auch eine Insel*. `http://www.galileocomputing.de/openbook/javainsel6/`.

[20] Wegener, Ingo: *Theoretische Informatik - eine algorithmenorientierte Einführung (2. Auflage)*. Teubner, 1999.

[21] World Wide Web, Consortium: *Document Object Model (DOM) Level 3 Core Specification*. `http://www.w3.org/TR/DOM-Level-3-Core/`.

[22] World Wide Web, Consortium: *Extensible Markup Language (XML) 1.1 (Second Edition)*. `http://www.w3.org/TR/xml11/`.

[23] World Wide Web, Consortium: *Extensible Stylesheet Language (XSL) Version 1.1*. `http://www.w3.org/TR/xsl11/`.

[24] World Wide Web, Consortium: *Scalable Vector Graphics (SVG) 1.1 Specification*. `http://www.w3.org/TR/SVG11/`.

[25] World Wide Web, Consortium: *Web Services Description Language (WSDL) 1.1*. `http://www.w3.org/TR/wsdl`.

[26] World Wide Web, Consortium: *World Wide Web Consortium*. `http://www.w3.org/`.

[27] World Wide Web, Consortium: *XHTML 1.0 The Extensible HyperText Markup Language (Second Edition)*. `http://www.w3.org/TR/xhtml1/`.

[28] World Wide Web, Consortium: *XML Path Language (XPath) 2.0*. http://www.w3.org/TR/xpath20/.

[29] World Wide Web, Consortium: *XML Schema*. http://www.w3.org/XML/Schema.

[30] World Wide Web, Consortium: *XQuery 1.0: An XML Query Language*. http://www.w3.org/TR/xquery/.

[31] World Wide Web, Consortium: *XSL Transformations (XSLT) Version 2.0*. http://www.w3.org/TR/xslt20/.