Institut für Architektur von Anwendungssystemen
Universität Stuttgart
Universitätsstraße 38
70569 Stuttgart
Germany

Studienarbeit Nr. 2403

Unifizierte Service-Komposition für ConDec

Siguang, Liang

Studiengang: Informatik

Prüfer: Prof. Dr. Frank Leymann

Betreuer: Katharina Görlach

begonnen am: 08.11.2012

beendet am: 10.05.2013

CR-Klassifikation: D.3.1, F.4.2

Kurzfassung

Heutzutage werden immer mehr Web Services erstellt und regelmäßig modifiziert. Es empfiehlt sich, mehrere Web Services zu verbinden, besonders wenn kein einzelner Web Service die Anforderungen der Benutzer zufrieden stellen kann. Komposition von Web Services ist ein Prozess von Konstrukt des komplexen Web Services aus „atomic" Web Services, um die spezifische Aufgabe zu erledigen, d.h. eine Reihenfolge von Web Services. Aber die Web Services werden in verschiedenen Sprachen und Plattformen implementiert. Wir hoffen, dass es die Abstraktion für verschiedenen Plattformen und Sprachen gibt, so dass die Aufrufe der Web Services immer gleich sein können. In dieser Studienarbeit wird versucht die Transformation von ConDec zu formalen Grammatiken zu implementieren, um die unifizierte Service-Komposition über ConDec zu ermöglichen. Die Service-Kompositionen werden mit Hilfe der „hohen" Beschreibungssprache ConDec modelliert. Die resultierende Grammatik wird in einer XML-Datei mit einem vorgegebenen XML-Schema gespeichert und soll bei einem entsprechenden Automaten ausgeführt werden. Darüber hinaus wird die Architektur des Transformationsprogramms beschrieben. Einige Herausforderungen in der Transformation von ConDec zu formalen Grammatiken werden auch ausführlich diskutiert.

Inhalt

Kurzfassung 1
1 Einleitung 4
2 Grundlagen 5
2.1 Web Services 5
2.1.1 Web Services Kompositionen 6
2.2 Deklarative Sprache 7
2.2.1 ConDec 9
2.2.2 LTL 11
2.3 Formale Grammatiken 12
2.4 Transformation 12
3 Implementierung 16
3.1 Architektur des Programms 20
3.2 Herausforderungen 24
3.2.1 Existence Templates 24
3.2.2 Relation Templates 25
3.2.3 Choice Templates 26
3.2.4 Branching von Restriktionen 27
3.3 Testfälle 27
4 Zusammenfassung 31
Anhang 32
A.1 Die finale Kompositionsgrammatik vom Beispiel „religion“ 32
A.2 Die finale Kompositionsgrammatiken vom Beispiel „medical“. 34
Literaturverzeichnis 39
Erklärung 41

1 Einleitung

Heutzutage verwenden die Kompanien Computer mehr und mehr, um ihre Geschäfte zu unterstützen. Es ist via Business-Prozess vorgeschrieben, wie die Geschäfte geschaffen werden. Die Applikationen unterstützen Business-Prozesse und müssen mit Business-Prozessen übereinstimmen (Applikation = Businessprozess + Businessfunktion). Die Änderungen bei der Ausführung von Business müssen baldmöglichst in Applikationen reflektiert werden. Ein Workflow ist eine Ausführung von Business-Prozess in einem „computing environment". Immer mehr Leute interessieren sich für Workflow-Technologie. Workflow ist die Basis von Business Process Management (BPM), während BPM als „the next step" nach der Workflow-Welle im neuzigsten Jahrzehnt betrachtet wird[19]. Weske, Aalst und Verbeek definieren den Begriff BPM: "*Supporting business processes using methods, techniques, and software to design, enact, control, and analyze operational processes involving humans, organizations, applications, documents and other sources of information*[18]." BPM reduziert menschliche Fehler und falsche Kommunikation, lässt Workflow mehr effektiv und effizient ausführen. Ein Business-Prozess ist eine Aktivität oder eine Reihe der Aktivitäten, die ein spezifisches Ziel erreichen können.

Die heutigen Betriebe und Organisationen interessieren sich für die Implementierung ihrer Business-Prozesse via Web Services. Aber die Web Services werden in unterschiedlichen Sprachen implementiert, deshalb wird die Abstraktion der Web Services für verschiedene Sprachen und Plattformen versucht. Dadurch hoffen wir, dass die Aufrufe der Web Services immer gleich sind. Wenn kein einzelner Web Service die Anforderungen der Benutzer zufrieden stellen kann, interessieren sich die Benutzer für die Service-Kompositionen. Zuerst werden Web Services und ihre Kompositionen im Abschnitt 2 vorgestellt. Dann werden eine deklarative Sprache ConDec und formale Grammatik kurz vorgestellt. Darüber hinaus wird die Transformation von ConDec zu formalen Grammatiken beschrieben. Die Transformation wird in Java programmiert. Danach wird es im Abschnitt 3 diskutiert, wie die Transformation implementiert wird. In diesem Abschnitt wird die Architektur des Transformationsprogramms auch beschrieben. Einige Herausforderungen für die Transformation werden diskutiert. Dann zwei Testfälle werden geprüft. Die resultierenden Grammatiken werden mit Hilfe eines entsprechenden Automaten ausgeführt. Der Zweck ist, dass zum Schluss die Transformationen weiterer Hochsprachen zu Kompositionsgrammatiken implementiert und mit Hilfe der gleichen Automatenklassen ausgeführt werden. Beispielsweise werden auch BPEL-Modelle mit Hilfe eines Automaten ausgeführt, obwohl BPEL eine imperative Beschreibungssprache, im Gegensatz zur deklarativen Beschreibungssprache ConDec ist. Der Assembler-ähnliche Charakter von Kompositionsgrammatiken bildet dabei die Grundlage für die Unifizierung. Das ist „unifizierte Service-Komposition". Der Abschnitt 4 ist die Zusammenfassung der Arbeit.

2 Grundlagen

Vor der Diskussion des Themas „unifizierte Service-Komposition für ConDec“ werden einige diesbezügliche Technologien und Modellierungssprachen zunächst vorgestellt. Im Abschnitt 2.1 werden zuerst Web Services und ihre Kompositionen vorgestellt. Dann wird der Begriff deklarative Sprache im Abschnitt 2.2 ausführlich beschrieben. Danach wird formale Grammatik im Abschnitt 2.3 kurz vorgestellt. Zum Schluss werden die Transformationen von ConDec zu formalen Grammatiken im Abschnitt 2.4 diskutiert.

2.1 Web Services

„Web services are a new breed of Web application. They are self-contained, self-describing, modular applications that can be published, located, and invoked across the Web. Web services perform functions, which can be anything from simple requests to complicated business processes. Once a Web service is deployed, other applications (and other Web services) can discover and invoke the deployed service. “ ----IBM web service tutorial[9]

Web Services werden als selbstständige, modulare Einheiten von Applikationslogik beschrieben, die die Business-Funktionalität zu den anderen Applikationen durch Internetverbindung anbieten[3]. Einige Standards von Web Services werden eingeführt, um die Kommunikation zwischen verschiedenen Systemen in unterschiedlichen Plattformen, Programmiersprachen und Modellen zu realisieren, wie Universal Discription, Discovery and Integration(UDDI)[4] , Web Services Discription Language(WSDL)[5] und Simple Object Access Protocol(SOAP)[6]. UDDI ist ein Verzeichnisdienst, der speziell für die dynamischen Aspekte der Katalogisierung von Web Services entworfen wurde. UDDI sucht nicht bei den Anbietern nach Services. UDDI stellt eine Schnittstelle zwischen Nutzern und Anbietern dar. Die Anbieter veröffentlichen ihre Services in UDDI, welche von den Nutzern durchsucht werden. SOAP ist eine Layout-Spezifikation von Nachrichten, mit deren Hilfe Daten zwischen Systemen ausgetauscht und Remote Procedure Calls (RPC) durchgeführt werden können. WSDL ist eine plattform-, programmiersprachen- und protokollunabhängige Beschreibungssprache für Web Services zum Austausch der Nachrichten auf Basis von XML. Web Services werden durch sechs XML-Hauptelemente definiert: Datentypen (*types*), Nachricht (*message*), Schnittstellentypen (*portType*), Bindung (*binding*), Port (*port*) und Service (*service*). Datentypen definieren die verwendeten Datentypen. Nachrichten definieren die zu übertragenden Nachrichten abstrakt. Schnittstellentypen setzen sich aus abstrakten Operationen zusammen und definieren damit eine abstrakte Schnittstelle. Binding verknüpft die abstrakte Schnittstelle an ein konkretes Datenformat und Protokoll. Port weist einem Binding-Element eine konkrete Adresse zu. Service bildet die nach außen zugänglichen Elemente eines Service durch mehrere Ports. Klient möchte einen bestimmten Web Service nutzen und benötigt hierfür die oben sechs Elemente. Diese Informationen werden mit WSDL genau beschrieben. WSDL funktioniert als eine zu veröffentlichende Schnittstellenbeschreibung. Nutzer vom Web Service kennt nur WSDL, aber braucht Programm-Code nicht zu kennen. Eine populäre Kategorie von Web Services ist „syntaktische Web Services“ und „semantische Web Services“[3]. Für den syntaktischen Forschungsansatz sind die Schnittstellen von Web Services ähnlich wie Remote Procedure Call (RPC), während der semantische Forschungsansatz auf logisches Denken über Web Ressourcen fokussiert. In der Arbeit sind die vorkommenden Web Services „syntaktisch“.

Die traditionellen Webseiten haben hauptsächlich zur Aufgabe Informationen zu sammeln. Aber durch Web Services kann die Software Applikationen auf Webseiten zugegriffen werden und ausgeführt werden. Web Services unterstützen die Interaktionen zwischen Business Partners und Business Prozessen dadurch, dass Web Services die Computer und Geräte untereinander verbinden und die Daten synchron oder asynchron durch Internet austauschen[9]. Immer mehr der heutigen Betriebe und Organisationen implementieren ihre Kerngeschäfte und outsourcen andere Applikation Services im Internet. Wegen der Vorteile von Web Services steigt das Interesse für diesen Vorgang immer mehr, sowohl in Forschung, als auch in der Industrie.

2.1.1 Web Services Kompositionen

Weil die Zahl der vorhandenen Web Services stetig zunimmt, steigt auch der Bedarf der Kompositionen der grundlegenden Web Services. Die Forschung über Service-Kompositionen wird populärer und viele Resultate erscheinen seit einigen Jahren in der Fachliteratur, z.B. Business Process Model Language (BPML) von Business Process Management Community und XML Process Definition Language (XPDL) von Workflow Community[10]. D. Skogan, R. Gronmo und I. Solheim meinen, dass obwohl einige Organisationen die Kompositionssprachen vorgeschlagen hätten, gäbe es aber bis jetzt noch keinen bestimmten Sieger[10]. Deshalb ist es ihre Intention, Web Services Kompositionen durch UML Aktivitätsmodelle zu entwerfen. Darüber hinaus verwenden sie OMG's (Object Management Group) Model Driven Architecture (MDA), um die ausführbaren Spezifikationen in verschiedenen Kompositionssprachen zu erzeugen. Es gibt zwei Schwerpunkte in ihrer Arbeit. Einer ist die Transformation von WSDL Beschreibung zu UML für die Fertigstellung von Kompositionsmodellen. Der andere Schwerpunkt ist die Unabhängigkeit von verschiedenen Web-Services-Kompositionssprachen. Wegen der Unabhängigkeit kann ein Benutzer seine vorgezogene Kompositionssprache auswählen. Ihre Theorie ist schon in zwei Kompositionssprachen BPEL4WS und WorkSCo geprüft geworden. H. Foster, S. Uchitel, J. Magee und J. Kramer beschreiben eine formale Methode mit Hilfe von Finite State Processes (FSP) Notation, die die Kompositionen von Web Services Workflows modellieren und verifizieren kann[7]. Im heutigen Web-Umfeld werden zahllose Web Services erzeugt und aktualisiert. J. Rao und X. Su meinen, dass die Analyse von Web Services und die manuelle Herstellung von Kompositionsplänen schon über die menschlichen Fähigkeiten hinausgehen. Sie lösen das Problem durch Cross-Enterprise Workflow und AI Planning[12].

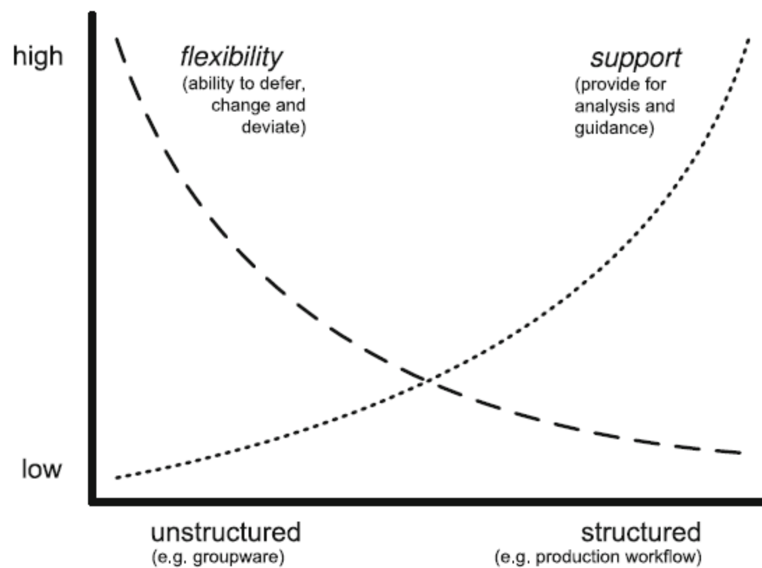
Durch die vorgestellten Forschungen kann deutlich werden, dass wegen der raschen Entwicklung von Web Services deren Kompositionen immer wichtiger werden. Es zeigt auch, dass die Web Services Kompositionen trotzdem eine sehr komplexe Herausforderung darstellen können, obwohl es schon viele Forschungen zu verschiedenen Aspekten gibt. Meiner Meinung nach verursachen drei Gründe diese Komplexität. Erstens, die Zahl von Web Services steigt zu schnell seit einigen Jahren. Jeder Benutzer kann ein riesiges Repository von Web Services nutzen. Das bedeutet, dass sich die Komplexität von Services-Kompositionen steigert. Zweitens, Web Services werden regelmäßig erzeugt und aktualisiert. Deshalb muss das Kompositionssystem die Aktualisierung während der Laufzeit suchen und finden. Sie trifft auch die Entscheidung über aktuelle Informationen. Drittens, Web Services werden von unterschiedlichen Organisationen entwickelt. Diese Organisationen verwenden unterschiedliche Konzept-Modelle, Sprachen und Plattformen, um die Web Services zu realisieren. Meines Wissens

nach gibt es noch keine eindeutige Beschreibungssprache, die Web Services in eine identische Bedeutung transferieren und bewerten kann. Die Forschungsrichtung dieser vorliegenden Arbeit soll den dritten Aspekt zum Gegenstand haben. Die Möglichkeit zur unifizierten Modellierung und Ausführung von Service-Kompositionen auf Basis von formalen Grammatiken und Automatentheorie wird diskutiert. Es ist beispielsweise möglich, obwohl BPEL eine imperative Sprache und ConDec eine deklarative Sprache ist, dass die beiden mit Hilfe der eingeführten Kompositionsgrammatiken durch denselben Automaten ausgeführt werden können. Die Studienarbeit fokussiert auf die Transformation von ConDec zu den eingeführten formalen Grammatiken.

2.2 Deklarative Sprache

Heutzutage ist der Begriff „imperative Sprache" für die Benutzer nicht mehr fremd, z.B. BPEL. Trotzdem werden die deklarativen Sprachen mehr und mehr angewendet, denn sie haben bessere Flexibilität. Der Unterschied zwischen imperativen und deklarativen Sprachen ist offensichtlich. Die Vorgehensweise von den imperativen Sprachen ist „*say how to do something*", während das Prinzip der deklarativen Sprachen „*say what is required and let the system determine how to achieve it*" ist[13]. Die Betrachtungsweise der *imperativen* Sprachen wird als „inside-to-outside" charakterisiert[14]. Es spezifiziert hauptsächlich die Prozeduren, wie die Arbeit getan wird. Die imperativen Sprachen erfordern, alle Alternativen im Modell vor der Ausführung der Prozesse explizit zu spezifizieren. Alle neue Alternativen müssen zu dem Modell während der „build-time" hinzugefügt werden. Aber einige Leute meinen, dass die imperative Betrachtungsweise übermäßig spezifiziert ist[2]. Im Gegensatz dazu verwendet die Vorgehensweise der *deklarativen* Sprachen eine „outside-to-inside" Methode[14]. Es spezifiziert die Prozesse nicht als „*apriori*". Die deklarative Vorgehensweise determiniert nicht, wie die Prozesse genau arbeiten, sondern nur die nötigen Eigenschaften werden beschrieben.

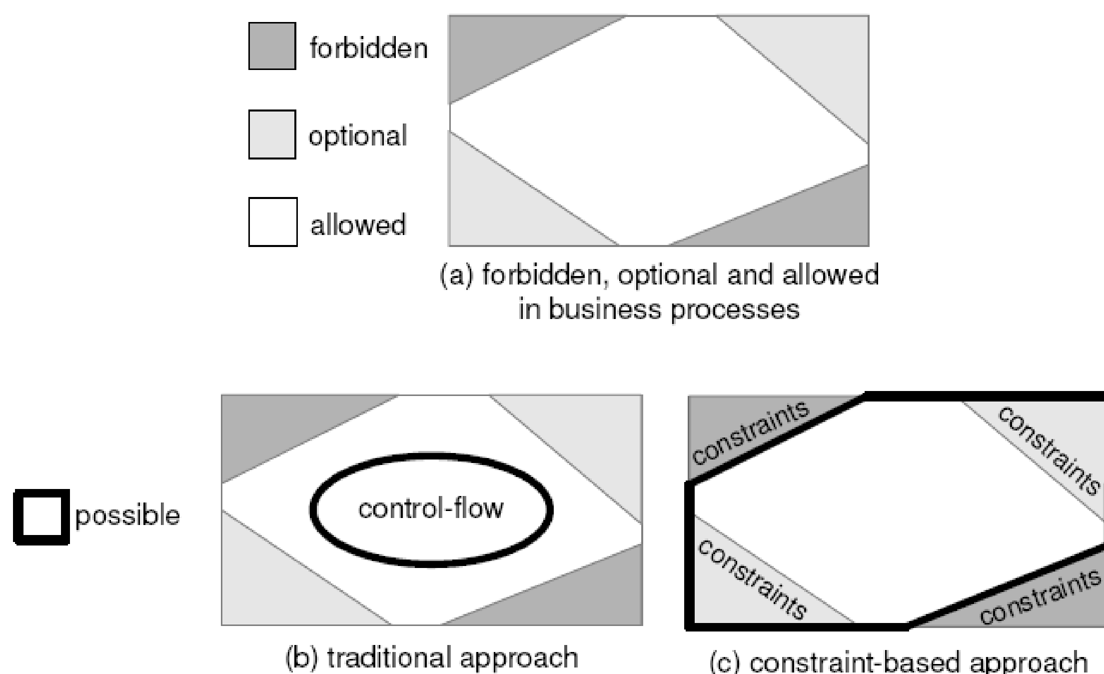
Die Entwicklung von Workflow Management System (WFMS) ist von dem Konflikt der Benutzeranforderungen beschränkt. Einerseits gibt es die Anforderung der Kontrolle aller Prozesse, so dass die unkorrekte oder unerwünschte Ausführung der Prozesse vermieden werden können. Andererseits möchten die Benutzer die flexibel Prozesse, die ihre Aktionen nicht einschränken. Einige Forscher meinen, dass WFMS zu restriktiv ist[15]. Die Abbildung 1 beschreibt die Paradoxie. Die Benutzer hoffen die hohe Flexibilität und die hohe Unterstützung. Aber hohe Unterstützung bedeutet enge Struktur von Systemen, d.h. niedrige Flexibilität. Je höher die Flexibilität ist, desto lockerer werden Systeme strukturiert, d.h. niedrige Unterstützung. Systeme bieten Unterstützung oder Flexibilität, aber nicht die beide. Die Benutzer brauchen die Balance zu finden.



Die Abbildung 1. Die Systeme bieten Unterstützung oder Flexibilität, aber nicht die beide. Die Benutzer brauchen die Balance zu finden[16].

Die deklarative Vorgehensweise kann die beiden Anforderungen zwischen der Unterstützung und Flexibilität gut ausgleichen. Die Abbildung 2 zeigt den Vergleich von traditioneller und „constraint-based" Vorgehensweise. Die Abbildung 2(a) zeigt drei Typen Szenarios in Business-Prozessen: (1) Verbotene (*forbidden*) Szenarios sollen nie geschehen, (2) Optionale (*optional*) Szenarios sind erlaubt, aber sollen bei den meisten Fällen vermieden werden. (3) Erlaubte (*allowed*) Szenarios können unbedenklich ausgeführt werden. Die Workflow-Management-Systeme definieren und führen die Modelle der Business-Prozesse aus, die die Reihenfolge der Aktivitäten in Business-Prozessen spezifizieren. In den traditionellen Workflow-Management-Systemen spezifizieren die Prozess-Modelle die Reihenfolge der Aktivitäten *explizit*, z.B. „*control-flow*" eines Business-Prozesses. Mit anderen Worten kann die Ausführung von Business-Prozessen nur gemäß der expliziten Spezifikation im „*control-flow*", wie die Abbildung 2(b) zeigt. Das schwarze Oval zeigt die Grenze der Aktivitäten, die vom traditionellen (z.B. imperativen) Prozess-Modell nach der Modellierungsmethode „inside-to-outside" definiert werden. Wegen der hohen Unberechenbarkeit der Business-Prozesse werden viele Ausführungen von erlaubten und optionalen Szenarios nicht vorher erwartet. Sie werden nicht explizit in *„control-flow"* inkludiert. Deshalb ist es für die traditionellen Systeme unmöglich, alle Untermengen von *erlaubten* Szenarios auszuführen, wie die Abbildung 2(b). Die Abbildung 2(c) zeigt die auf Restriktionen basierende (*„constraints-based"*) Vorgehensweise, die die beiden *erlaubten* und *optionalen* Szenarios ausführen kann. Statt der expliziten Spezifikation, was *möglich* in Business-Prozess ist, spezifizieren die auf Restriktionen basierenden Modelle, was *verboten* ist. Die mögliche Ordnung der Aktivitäten wird durch die Restriktionen implizit spezifiziert. Darüber hinaus gibt es zwei Typen Restriktionen: (1) Mandatorische (*mandatory*) Restriktionen fokussieren auf die verbotenen Szenarios, (2) Optionale (*optional*) Restriktionen spezifizieren die optionalen Szenarios. Irgendetwas, das die mandatorischen Restriktionen nicht verstößt, ist in der Ausführung eines Modells möglich. Die deklarative Betrachtungsweise ist auch auf Restriktionen basierend. Die schwarze fette Grenze in der Abbildung 2(c) repräsentiert die Modellierungsmethode „*outside-to-inside*", die von der deklarativen Vorgehensweise unterstützt wird. Die deklarative Vorgehensweise ermöglicht die Flexibilität ohne den Verlust von Unterstützung. Einerseits bietet die deklarative Vorgehensweise mehrere Möglichkeiten

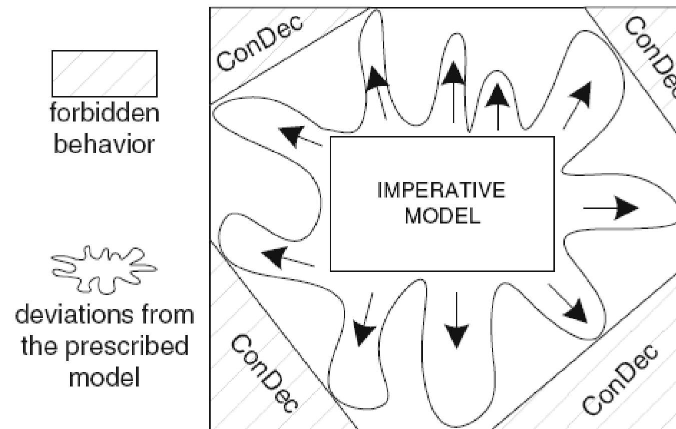
der Ausführung von Modell als die imperative Vorgehensweise. Die Benutzer können die lokale Entscheidung machen, wie der Business-Prozess ausgeführt wird. Andererseits können die Benutzer die mehreren Restriktionen im deklarativen Prozess-Modell verfolgen, sodass die Benutzer diese Restriktionen nicht verstoßen. Darüber hinaus zeigt die deklarative Vorgehensweise den Unterschied zwischen mandatorischen Restriktionen (*must be followed*) und optionalen Restriktionen (*should be followed*). Beim ersten Fall dürfen die Benutzer die Restriktionen gar nicht verstoßen. Beim zweiten Fall können die Benutzer die Restriktionen verstoßen. Aber die Benutzer werden im Voraus gewarnt. Die Abbildung 2 zeigt den Unterschied zwischen imperativer und deklarativer Vorgehensweise deutlich. Die traditionelle (z.B. imperative) Vorgehensweise verwendet die Verfahrensprozess-Modelle, um die Ausführungsprozesse explizit (d.h. step-by-step) zu spezifizieren. Die deklarative Betrachtungsweise ist auf „constraints" (Restriktionen) basierend. *„Anything is possible as long as it is not explicitly forbidden*[16].*"* Deshalb spezifizieren die auf Restriktionen basierenden Modelle die Ausführungsprozesse implizit mit Hilfe von Restriktionen, solang keine Ausführung die Restriktionen verstößt. Die deklarative Vorgehensweise kann die Balance zwischen die Flexibilität und Unterstützung finden.



Die Abbildung 2. Der Vergleich zwischen der traditionellen(z.B. imperativ) und der auf Restriktionen basierenden(z.B. deklarativ) Vorgehensweise[2]

2.2.1 ConDec

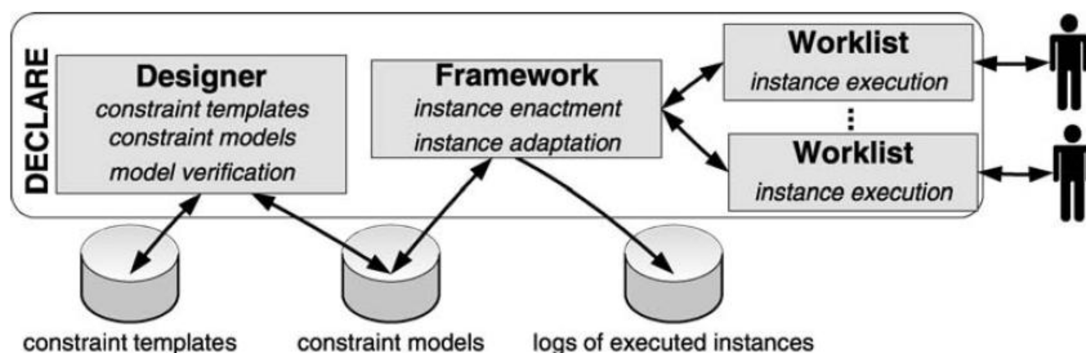
ConDec ist eine deklarative, graphische Sprache, die von Aalst und Pesic[17] im Forschungsgebiet von Business Process Management (BPM) vorgeschlagen wird. Es zielt auf die Spezifikation, Konstruktion und Überwachung von Business Prozessen mit Hilfe von Restriktionen zwischen Aktivitäten. Jetzt wird ConDec auch im Gebiet Service-Oriented Architecture (SOA) angewandt. Die Abbildung 3 zeigt den Unterschied zwischen imperative Sprachen und ConDec. ConDec beginnt mit allen Möglichkeiten („what") und nähert sich dem erwünschten Verhalten (outside-to-inside). Imperative Sprachen beginnen mit der expliziten Spezifikation von Prozessen („how") und detaillieren die Prozesse ausführlich (inside-to-outside).



Die Abbildung 3. Deklarative Sprache ConDec (outside-to-inside) vs. Imperative Sprachen (inside-to-outside)[17]

Ein ConDec Modell besteht hauptsächlich aus zwei Teilen: Aktivitäten und Beziehungen. Die Aktivitäten repräsentieren die Einheiten der Arbeit. Die Beziehungen beschreiben, wie die Aktivitäten ausgeführt werden, und werden als Restriktionen (*„constraints“*) genannt. In der folgenden Diskussion repräsentieren alle Aktivitäten „Web Services“.

In der Studienarbeit werden alle ConDec Modelle durch die Software „Declare“ konstruiert. Declare[1] ist ein auf Restriktionen basierendes WFMS, das ConDec sowie andere deklarative Sprachen, wie DecSerFlow, unterstützt. Die Abbildung 4 zeigt die Architektur vom Declare System. Das System besteht aus drei Komponenten: Designer, Framework und Worklist. Die Aufgabe von Designer ist „Constraint Templates“ zu herstellen, so dass die konkreten Prozessmodelle konstruiert werden. Framework ist für die Ausführung der Beispiele von Prozessmodellen. Durch Worklist können die aktiven Instanzen erreicht werden.

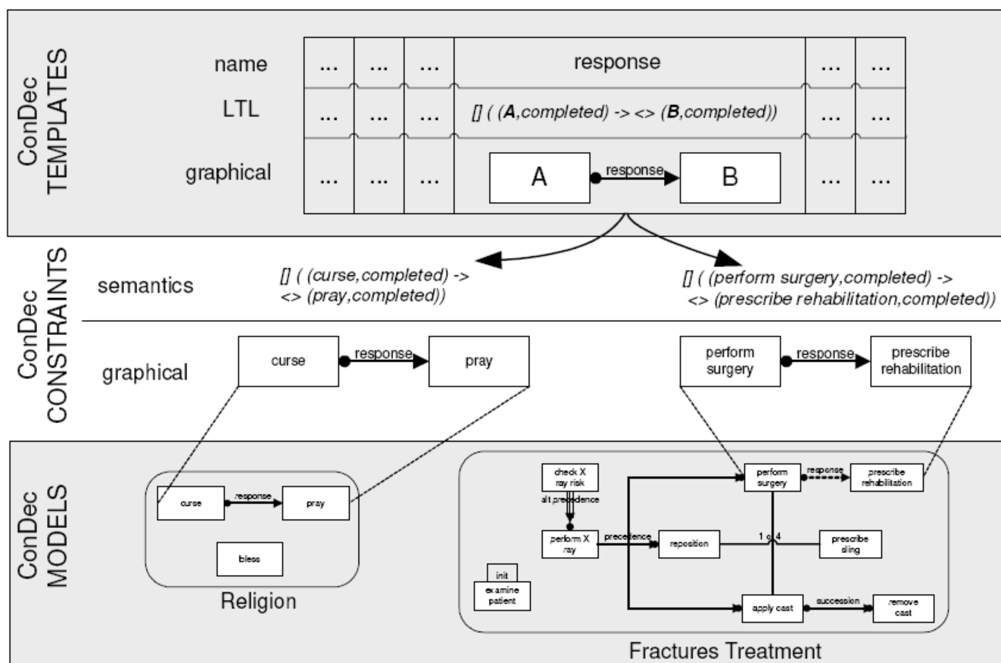


Die Abbildung 4. Die Architektur von Declare[16]. Es besteht aus 3 Komponenten: Desinger, Framework und Worklist.

Die meisten Forschungsansätze bieten eine Serie von vordefiniertem Konstrukt, die Abhängigkeit zwischen Aktivitäten in Prozess-Modellen zu beschreiben (z.B. *sequence, choice, pa-*

[1] http://www.win.tue.nl/declare/2011/11/declare-2-2-0-with-modules-released/

rallelism, loops usw.). Aber Declare ist anders. Declare benutzt eine „*customizable*" Serie von beliebigen Templates, die Restriktionstemplates („*constraint templates*") heißen. In Declare können die beliebigen auf Restriktionen basierenden Sprachen definiert werden. Für jede Sprache können die beliebigen Templates herstellt werden[2]. Es ist auch der Grund, warum Declare mehrere deklarativen Sprachen unterstützen kann. ConDec kann als eine Sammlung von „Constraint Templates" angesehen werden. Jedes Template hat (1) einen eindeutigen Namen, (2) eine graphische Repräsentation und (3) eine formale Spezifikation über seine Semantik, nämlich Linear Temporal Logic (LTL). Die Abbildung 5 zeigt, wie die Restriktionen durch die Templates in ConDec Modell herstellt werden. Die LTL Formel und die graphische Repräsentation werden für jede Restriktion nicht separat spezifiziert, sondern eine Restriktion ist auf ein Template basierend. Eine Restriktion erbt den Namen, die graphische Repräsentation und die LTL Formel aus dem entsprechenden Template. Die Restriktion wird in ConDec Modell graphisch repräsentiert, während die darunterliegende LTL Formel verhohlen bleibt.



Die Abbildung 5. ConDec Templates, Restriktionen und Modelle. Die graphische Repräsentation wird erbt, aber die LTL Formel ist verhohlen[2].

2.2.2 LTL

LTL ist ein Typ von Logik und für die Beschreibung der Sequenzen von Transitionen zwischen Zuständen in reaktiven Systemen. Wegen ihrer deklarativen Natur wird LTL auch verwendet, um die Restriktionen in den auf Restriktionen basierenden Service-Kompositionen zu spezifizieren. Zusätzlich von traditionellen logischen Operatoren (wie UND, ODER usw.) benutzt LTL auch temporale Operatoren, wie *always* ($\square$), *eventually* ($\Diamond$), *next* ($\circ$), *until* ($\cup$) und *weak until* (W)[2]. Z.B. für das Template *response(A, B)* wird seine Semantik in der LTL Formel $\square(A \Rightarrow \Diamond(B))$ gegeben. Das bedeutet „*Whenever activity A is executed, activity B has to be eventually executed afterwards*". ConDec ist auf Restriktionen basierend und LTL spezifiziert die Semantik formell. Weil LTL Formeln für Laien schwierig zu verstehen sind, ver-

bindet ConDec eine graphische Repräsentation zu jedem Template. Deshalb brauchen die Benutzer keine Kenntnisse über LTL, um ConDec zu benutzen. In der Arbeit wird der Unterschied von einigen Templates durch ihre LTL Formeln diskutiert.

2.3 Formale Grammatiken

In der Studienarbeit wird versucht die Transformation von ConDec zu formalen Grammatiken zu implementieren. Alle ConDec Templates und ihre Kompositionen werden im Format formaler Grammatiken umgewandelt. Die formalen Grammatiken sind mathematische Modelle von Grammatiken, die durch die formalen Sprachen beschrieben und erzeugt werden können. In der Arbeit sind die formalen Grammatiken für ConDec 4-Tupel $G=(V, \sum, P, S)$. V ist die Menge von Non-Terminals, z.B. S_1 und A_1. $\sum$ ist die Menge von Terminals, z.B. a und b. In P werden alle Produktionsregeln gespeichert. S ist das Startsymbol. Die eingeführten Grammatiken in der Arbeit fokussieren nur auf die Produktionsregeln, sondern nicht auf die Ordnung der Symbole[1]. Z.B. bedeutet die Regel $D_1 D_2 X \rightarrow x Y$ nicht, dass D_1 und D_2 sich auf der linken Seite vom Symbol X. Sie spezifiziert, dass D_1 und D_2 nötig sind, um x aus X zu produzieren. D_1 und D_2 sind im Kontext von X. Aber $D_1 X D_2 \rightarrow x Y$, $D_2 X D_1 \rightarrow x Y$, $X D_1 D_2 \rightarrow x Y$ und $X D_2 D_1 \rightarrow x Y$ sind semantisch equivalent. In den formalen Grammatiken für ConDec haben alle Non-Terminals „Typen“. Z.B. $A_i, B_i, C_i \in$ Services und $S_i \in$ Helpers.

Die in der Arbeit resultierenden Grammatiken werden mit Hilfe eines entsprechenden Automaten ausgeführt. Der Automat ist eine abstrakte Maschine, die ein Modell eines digitalen, zeitdiskreten Rechners ist. Der Automat kann die resultierenden Grammatiken einlesen. Nach der Ausführung kann der Automat ein „Wort“ ausdrucken, das die Spur der Ausführung spezifiziert.

2.4 Transformation

Die grammatischen Produktionsregeln (grammatical production rules) für die Templates in ConDec repräsentieren die Abhängigkeiten zwischen Aktivitäten, die in deklarativen Workflow Modellen spezifiziert werden. Vor allem beginnen die Produktionsregeln für die deklarativen Service-Kompositionen mit einem Start-Symbol, nämlich S_1 (falls kein *init()* und *strong init()* existiert) oder S_0 (falls es mindestens ein *init()* oder *strong init()* gibt). Das Start-Symbol hat mehrere Versionen, d.h. S_2, S_3... usw. Alle $S_i \in$ Helpers. In der Grammatik von *init()* oder *strong init()* gibt es $H_i \in$ Helpers. Außer S_i und H_i präsentieren alle anderen Non-Terminals die Aktivitäten im deklarativen Modell, z.B. A_i, B_i, C_i usw. In ConDec sind alle Aktivitäten Services, z.B. A_i, B_i, C_i ... $\in$ Services. Die Restriktionen für die deklarativen Service-Kompositionssprachen brauchen nichtdeterministische, kontextfreie (falls kein *init()* und *strong init()* existiert) oder kontextsensitive (falls es mindestens ein *init()* oder *strong init()* gibt) Produktionsregeln und auch das leere Zeichen „ε“, um den Prozess zu beenden, z.B. $S \rightarrow \varepsilon$.

Die angebotenen Templates in ConDec werden in vier Kategorien klassifiziert, d.h. Existence Templates, Relation Templates, Choice Templates und Negation Templates. Darüber hinaus gibt es auch eine Situation „branching of constraints“. Die Existence Templates sind unäre Restriktionen, z.B. *existence(A)*, *init(A)* und *absence(A)*. Das bedeutet, dass die betreffenden Regeln die Ausführung von Aktivität A beschränken.

Die Relation Templates sind binäre Restriktionen, die die Abhängigkeiten zwischen zwei Aktivitäten beschreiben. Z.B. beschreibt *response(A,B)*, dass die Aktivität B in die Zukunft

ausgeführt werden muss, wenn die Aktivität A mindestens einmal ausgeführt wird. Die Abbildung 6 zeigt die Produktionsregeln für die Restriktion *response(A,B)*. Falls A ausgeführt wird ($S_1 \rightarrow A_1$ und $A_1 \rightarrow a\ S_2$), muss B ausgeführt werden, um die Prozedur zu beenden ($S_2 \rightarrow B_2$, $B_2 \rightarrow b\ S_1$ und $S_1 \rightarrow \varepsilon$).

$$
\begin{array}{lll}
S_1 \rightarrow A_1 | B_1 | C_1 | \varepsilon & & \text{with: } A_i, B_i, C_i \in \text{Services} \\
A_1 \rightarrow aS_2 & & S_i \in \text{Helpers} \\
B_1 \rightarrow bS_1 & & a, b, c \in \Sigma \\
C_1 \rightarrow cS_1 & & \\
S_2 \rightarrow A_2 | B_2 | C_2 & & \\
A_2 \rightarrow aS_2 & & \\
B_2 \rightarrow bS_1 & & \\
C_2 \rightarrow cS_2 & &
\end{array}
$$

Die Abbildung 6. Produktionsregeln von Restriktion *response(A,B).*

Die Negation Templates beschreiben die negierten Versionen von einigen Restriktionen, aber sind nicht gleich der logischen Negation. Z.B. bedeutet *not responded existence(A,B)*, dass die Aktivität B darf nicht ausgeführt sein (vor und nach A), falls die Aktivität A ausgeführt ist. Die Abbildung 7 zeigt die Produktionsregeln von *not responded existence(A,B).* $A_1 \rightarrow aS_2$ und $B_1 \rightarrow bS_3$ garantieren die Restriktion. Es gilt zu beachten, dass die Restriktion *not co-existence(A,B)* die gleiche Abhängigkeit in den beiden Richtungen zwischen A und B beschreibt, d.h. A darf nicht ausgeführt sein, falls B ausgeführt ist, und die Umkehrung ist auch richtig. Weil die beiden Restriktionen gleich sind, sind ihre Regeln auch gleich.

$$
\begin{array}{lll}
S_1 \rightarrow A_1 | B_1 | C_1 | \varepsilon & & \text{with: } A_i, B_i, C_i \in \text{Services} \\
A_1 \rightarrow aS_2 & & S_i \in \text{Helpers} \\
B_1 \rightarrow bS_3 & & a, b, c \in \Sigma \\
C_1 \rightarrow cS_1 & & \\
S_2 \rightarrow A_2 | C_2 | \varepsilon & & \\
A_2 \rightarrow aS_2 & & \\
C_2 \rightarrow cS_2 & & \\
S_3 \rightarrow B_3 | C_3 | \varepsilon & & \\
B_3 \rightarrow bS_3 & & \\
C_3 \rightarrow cS_3 & &
\end{array}
$$

Die Abbildung 7. Produktionsregeln von *not responded existence(A,B)* und *co-existence(A,B).*

Die Choice Templates beschreiben die Auswahl zwischen einigen Aktivitäten. Z.B. *choice 1 of 3(A,B,D)* bedeutet, dass mindestens ein Aktivität (von A, B, D) ausgeführt wird. *Exclusive choice 2 of 3(A,B,D)* bedeutet, dass nur zwei Aktivitäten (aus A, B, D) ausgeführt sein dürfen.

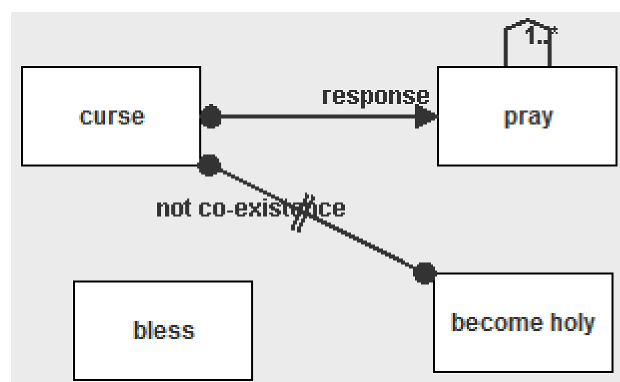
Die oben erwähnten Templates können erweitert sein. Z.B. *response(A,B)* kann zu *response*$((A_1,A_2),B)$, *response*$(A,(B_1,B_2))$ oder *response*$((A_1,A_2),(B_1,B_2,B_3))$ erweitert werden. In ConDec heißt diese Erweiterung „branching". Die Abbildung 8 ist die Regeln für *response*$(A,(B_1,B_2))$. Die Aktivität B besteht aus B_1 und B_2. Der Algorithmus von „branching constraints" wird in der Arbeit[1] referenziert. Weil die Indexe von Non-Terminals z.B. B_{11}

im Transformationsprogramm nicht so repräsentiert werden können, wie in der Abbildung 8, müssen die Non-Terminals im Programm verbessert werden. Statt B_{11} wird B(1,1) angewandt. Das erste „1“ ist der erste Index „$_1$“ von B_{11}, während das zweite „1“ von B(1,1) ist der zweite Index „$_1$“. Ähnlich statt B_{21} wird B(2,1) angewandt, usw. Natürlich werden die Non-Terminals mit nur einem Index, wie S_1, A_1, auch verbessert, um mit die Non-Terminals wie B(1,1), B(2,1) übereinzustimmen. Statt S_1 wird S(1) benutzt. Ähnlich statt A_1, B_1, C_1 werden A(1), B(1), C(1) im Programm angewandt.

$$
\begin{array}{lll}
S_1 \rightarrow A_1 \mid B_{11} \mid B_{21} \mid C_1 \mid \varepsilon & & \text{with: } A_i, B_i, C_i \in \text{Services} \\
A_1 \rightarrow aS_2 & & S_i \in \text{Helpers} \\
B_{11} \rightarrow b_1S_1 & & a, b, c \in \Sigma \\
B_{21} \rightarrow b_2S_1 & & \\
C_1 \rightarrow cS_1 & & \\
S_2 \rightarrow A_2 \mid B_{12} \mid B_{22} \mid C_2 & & \\
A_2 \rightarrow aS_2 & & \\
B_{12} \rightarrow b_1S_1 & & \\
B_{22} \rightarrow b_2S_1 & & \\
C_2 \rightarrow cS_2 & &
\end{array}
$$

Die Abbildung 8. Die Produktionsregeln von *response(A,(B_1,B_2))*.

In allen Produktionsregeln für die ConDec Restriktionen hat das Non-Terminal C spezielle Bedeutung. C bedeutet „alle andere Aktivitäten“. Z.B. in der Regeln für *response(A,B)* bedeutet C alle andere Aktivitäten außer den Aufgaben A und B. Falls es in einer XML-Datei nur eine Restriktion *response(A,B)* gibt, können wir das Non-Terminal C in den Produktionsregeln einfach anwenden, wie die Abbildung 6 zeigt. Falls es mehrere Restriktionen in der XML-Datei gibt, ist das Non-Terminal C komplexer. Z.B. ist die Abbildung 9 ein ConDec Modell „*religion*“. Es hat vier Aktivitäten und drei Restriktionen. Für die Restriktion *existence(pray)* bedeutet C (alle andere Aktivitäten) „curse“, „bless“ und „become holy“. Für die Restriktion *response(curse,pray)* wird C durch „bless“ und „become holy“ in seinen Produktionsregeln ersetzt. Für *not co-existence(curse, become holy)* repräsentiert C „pray“ und „bless“ . Die Abbildung 10 ist die Produktionsregeln von *response(curse,pray)*. In der Regeln ist C „bless“ und „become holy“. Die Anwendung von Non-Terminal C in anderen Restriktionen *existence(pray)* und *not co-existence(curse, become holy)* ist analog. Das bedeutet, dass das Non-Terminal C für verschiedene Restriktionen in einer XML-Datei unterschiedlich ist.



Die Abbildung 9. Ein ConDec Modell „religion“

$S_1 \rightarrow curse_1 | pray_1 | bless_1 | become.holy_1 | \varepsilon$ with: $curse_i, pray_i, bless_i, become.holy_i \in Services$
$curse_1 \rightarrow curse_t\ S_2$ $S_i \in Helpers$
$pray_1 \rightarrow pray_t\ S_1$ $curse_t, pray_t, bless_t, become.holy_t \in \Sigma$
$bless_1 \rightarrow bless_t\ S_1$
$become.holy_1 \rightarrow become.holy_t\ S_1$
$S_2 \rightarrow curse_2 | pray_2 | bless_2 | become.holy_2$
$curse_2 \rightarrow curse_t\ S_2$
$pray_2 \rightarrow pray_t\ S_1$
$bless_2 \rightarrow bless_t\ S_2$
$become.holy_2 \rightarrow become.holy_t\ S_2$

Die Abbildung 10. Die Produktionsregeln von *response(curse,pray)* im Modell „religion".

In den durch das Programm produzierten Regeln, werden die Namen von Aktivitäten einfach statt den A, B, C usw. verwendet, wie die Abbildung 10. Hierbei kann ein Problem entstehen. Z.B. im oben erwähnten Restriktion *response(curse,pray)* beinhaltet der Name von der Aktivität „become holy" ein Leerzeichen. Als Name von Aktivität in ConDec Modell ist diese Situation erlaubt. Aber das Programm benutzt den Namen von Aktivität einfach als den Namen von Non-Terminal. Deshalb wird „become holy" als der Name von Non-Terminal in den Produktionsregeln auftreten. Weil die produzierten Regeln durch einen Automaten ausgeführt werden müssen und der Automat das Leerzeichen als Trennungszeichen von Symbolen in den Produktionsregeln behandelt, müssen die Namen von Non-Terminals mit Leerzeichen verbessert werden. Das Leerzeichen in Namen von Non-Terminals werden durch ein Punkt „." ersetzt. Z.B. für den Aktivitätsnamen „become holy" ist sein entsprechender Non-Terminalname in den Produktionsregeln „become.holy". Die Terminals sollen auch verbessert werden. Weil die Non-Terminals A, B, C usw. durch die Namen von Aktivitäten ersetzt werden, werden ihre entsprechenden Terminals a, b, c usw. auch durch „Non-Terminalname_t" ersetzt. Z.B. für das Non-Terminal „become.holy" ist „become.holy_t" sein entsprechendes Terminal in den Produktionsregeln, wie in der Abbildung 10.

3 Implementierung

Im Abschnitt wird es ausführlich vorgestellt, wie das Transformationsprogramm die Transformation implementiert. Im Abschnitt 3.1 werden die Architektur des Programms durch das Klassendiagramm und das Aktivitätsdiagramm beschrieben. Im Abschnitt 3.2 werden einige Ausforderungen diskutiert. Die Inhalte von Abschnitt 3.3 sind über zwei Testfälle.

Wenn ein graphisches Modell in der Software Declare gespeichert wird, wird es in einer XML-Datei gespeichert. Meine Aufgabe ist, ein Transformationsprogramm zu erstellen. Die Eingabedatei ist die auf ConDec basierende XML-Datei, und die Ausgabedatei ist eine Kompositionsgrammatik, die durch einen Automaten ausgeführt werden kann. Eine solche XML-Datei von ConDec beinhaltet viele Elemente, die die graphische Repräsentation beschreiben, z.B. <cellheight="**50.0**" id="**1**" width="**90.0**" x="**179.0**" y="**155.0**" />. Diese Informationen sind für die Studienarbeit nicht relevant und werden daher nicht weiter behandelt. Das Transformationsprogramm zieht nur die notwendigen Informationen über die Restriktionen heraus, d.h. die Informationen, die sich in Tag-Paaren <acitivitydefinitions>...</acitivitydefinitions>, <data>...</data> und <constraintdefinitions>...</constraintdefinitions> befinden. Aber nicht alle Informationen in diesen Tag-Paaren sind nötig.

```
<activitydefinitions>
  <activity id="1" name="kochen 1">
    <authorization>
       <teamrole>1</teamrole>
    </authorization>
    <datamodel>
      <data element="1" type="0" />
      <data element="2" type="1" />
    </datamodel>
  </activity>
......
  <activity id="5" name="essen 5">
    <remote task="Hunger 6" />
    <authorization />
    <datamodel>
      <data element="3" type="0" />
      <data element="6" type="1" />
      <data element="2" type="0" />
    </datamodel>
  </activity>
......
</activitydefinitions>
```

Die Abbildung 11. <activitydefinitions> beinhaltet alle Aktivitäten, die in XML-Datei vorkommen.

Das Element <activitydefinitions>...</activitydefinitions> beinhaltet alle Aktivitäten in der XML-Datei, z.B. <activity id="**1**" name="**kochen 1**">...</activity>, <activity id="**5**" name="**essen 5**">...</activity> usw., wie die Abbildung 11. Die Eigenschaft „name" ist der Name von Aktivität. Die Namen dürfen nicht dupliziert sein. Jedes Tag (Element in spitzen Klammern eingeschlossenen Kürzel) <activity> kann das Tag <remote task="XXX"> und das Tag <datamodel> beinhalten (<autorization> hat keine Beziehung zur Arbeit.). <remote task="XXX"> bedeutet, dass diese Aktivität ein Web Service repräsentiert. <datamodel> zeigt die Ein/Ausgabe-Parameters der Aktivität, z.B. <data element="**1**" type="**0**" />. In der XML-Datei gibt es noch das Tag <dataelement id="**1**" initial="**aa**" name="**X**" type="**string**" />, wie die Abbildung 12. Die Bedeutungen von der Eigenschaft id="1" und der Eigenschaft element="1" in <data element="**1**" type="**0**" /> sind gleich. Zuerst wird die Eigenschaft element="1" in <data element="**1**" type="**0**" /> herausgezogen. Dann suchen wir in <data>...</data>, welches <dataelement id="XX" initial="XX" name="XX" type="XX"> die Eigenschaft id="1" hat. Falls ein solches <dataelement> gefunden ist, werden <data element="**1**" type="**0**" /> und <dataelement id="**1**" initial="**aa**" name="**X**" type="**string**" /> verbunden. Durch element="1" und id="1" können die Initialisierungsinhalte, Name und Typ vom Ein/Ausgabe-Parameter gefunden werden. Die Eigenschaft type="0" von <data> in <datamodel> in der Abbildung 11 zeigt, ob es Eingabe oder Ausgabe oder Eingabe/Ausgabe ist. Type="0" bedeutet Eingabe, type="1" bedeutet Ausgabe, type="2" bedeutet Eingabe/Ausgabe. Durch <data element="**1**" type="**0**" /> in der Abbildung 11 und <dataelement id="**1**" initial="**aa**" name="**X**" type="**string**" /> in der Abbildung 12 können wir wissen, dass die Aktivität „**kochen 1**" eine Eingabe hat. Der Eingabename ist „**X**". Der Eingabetyp ist „**string**". Der Initialisierungsinhalt ist „**aa**".

```
......
<data>
  <dataelement id="1" initial="aa" name="X" type="string" />
  <dataelement id="2" initial="bbb" name="Y" type="string" />
  <dataelement id="3" initial="5" name="Z" type="integer" />
  <dataelement id="4" initial="0.0" name="U" type="double" />
  <dataelement id="5" initial="7" name="V" type="integer" />
  <dataelement id="6" initial="true" name="W" type="boolean" />
</data>
......
```

Die Abbildung 12. <dataelement> beinhaltet die Eigenschaften von Ein/Ausgabe-Parameters.

Das Element <constraintdefinitions>...</constraintdefinitions> umfasst alle Restriktionen in der XML-Datei, wie in der Abbildung 13. Es beinhaltet ein oder mehrere <constraint>-Elemente. Jedes <constraint> repräsentiert ein entsprechendes Template und beinhaltet ein Element <template>. Jedes <template> hat ein paar Tags, z.B. <name>**response**</name>. Das ist der Typ von Template. Dadurch kann das in Java programmierte Transformationsprogramm entscheiden, welche Grammatik zu produzieren ist. In <text>...</text> ist die LTL-Formel von Template. In <parameters>...</parameters> werden ein oder mehrere <parameter>-Elemente inkludiert. Jedes <parameter> hat drei Attribute: „branchable", „id", „name", z.B. für die Restriktion „*response(A, B)*" gibt es <parameter branchable="true" id="2" na-

me=“A“>...</parameter> und <parameter branchable=“true“ id=“1“ name=“B“>...</parameter>, wie in der Abbildung 13. Branchable=“true“ bedeutet, dass die Aktivität „branchable“ sein kann. „id“ verbindet <parameter> in <parameters> mit <parameter> in <constraintparameters>. „name“ zeigt den Namen der abstrakten Aktivität in LTL, sondern nicht den wirklichen Namen der Aktivität im Template. Wir berücksichtigen nur die Attribute „id“ und „name“. In <constraintparameters> werden die wirklichen Namen der Aktivitäten im Template gespeichert. <constraintparameters> beinhaltet ein oder mehrere <parameter templateparameter=“X“>. Die Attribut templateparameter=“X“ wird als die Verbindung mit <parameter branchable=“X“ id=“X“ name=“X“> benutzt. Jedes <parameter templateparameter=“X“> inkludiert ein <branches>. Jedes <branches> inkludiert ein oder mehrere <branch name=“XXX“>. <branch name=“XXX“> speichert den wirklichen Namen der Aktivität. Z.B. für <parameter branchable=“true“ id=“2“ name=“A“> in der Abbildung 13 bedeutet name=“A“ die abstrakte Aktivität A in LTL []((“A“ --> <>(“B“))). Branchable=“true“ bedeutet, dass die abstrakte Aktivität A „branchable“ sein kann. Das Programm zieht id=“2“ heraus. Dann sucht das Programm <parameter templateparameter=“2“> in <constraintparameters>. Falls es gefunden wird, werden alle Attribute name=“XXX“ aus <branch name=“XXX“> herausgezogen. In der Abbildung 13 bekommen wir < branch name=“**kochen 1**“/> und < branch name=“**kochen 2**“/>. Das bedeutet, dass die abstrakte Aktivität A aus zwei wirkliche Aktivitäten „**kochen 1**“ und „**kochen 2**“ besteht, d.h. *response((**kochen 1**, **kochen 2**), B)*. Ähnlich kann die abstrakte Aktivität B analysieren.

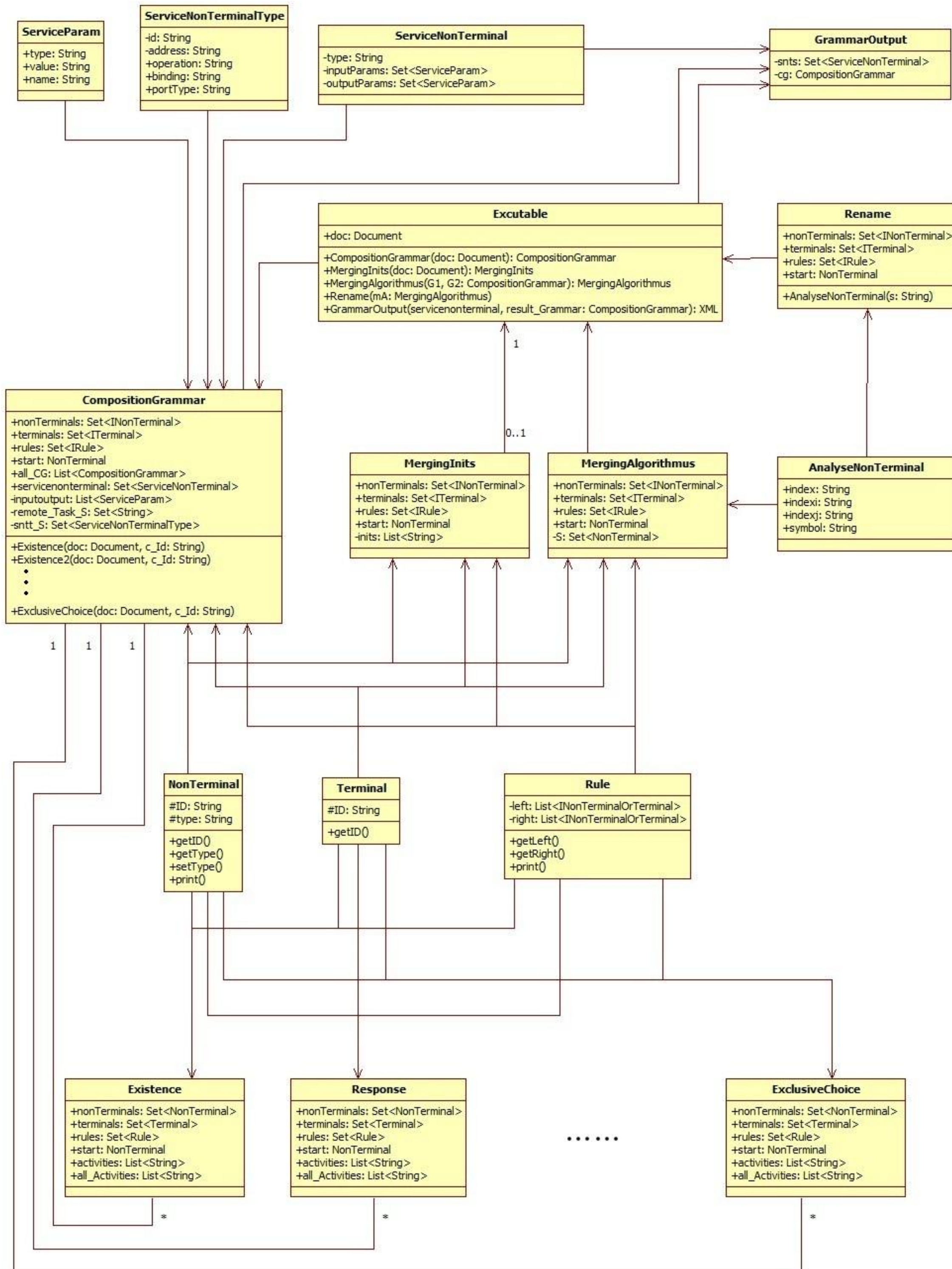
```
<constraintdefinitions>
    <constraint id="4" mandatory="true">
        ......
        <template>
        ......
          <name>init</name>
          <text>( ( "A.started" \/ "A.cancelled" ) W "A" )</text>
          <parameters>
              <parameter branchable="false" id="1" name="A">
              ......
              </parameter>
          </parameters>
          ......
        </template>
        <constraintparameters>
           <parameter templateparameter="1">
               <branches>
                   <branch name="Hurger 6" />
               </branches>
           </parameter>
        </constraintparameters>
    </constraint>
    <constraint id="5" mandatory="true">
    ......
        <template>
        ......
          <name>response</name>
          <text>[]( ( "A" -> <>( "B" ) ) )</text>
          <parameters>
              <parameter branchable="true" id="2" name="A">
              ......
              </parameter>
              <parameter branchable="true" id="1" name="B">
              ......
              </parameter>
          </parameters>
          ......
        </template>
        <constraintparameters>
           <parameter templateparameter="2">
               <branches>
                   <branch name="kochen 1" />
                   <branch name="kochen 2" />
               </branches>
           </parameter>
           <parameter templateparameter="1">
               <branches>
                   <branch name="essen 3" />
                   <branch name="essen 4" />
                   <branch name="essen 5" />
               </branches>
           </parameter>
        </constraintparameters>
     </constraint>
     ......
</constraintdefinitions >
```

Die Abbildung 13. <constraintdefinitions> inkludiert alle Restriktionen von XML-Datei.

3.1 Architektur des Programms

Die Abbildung 14 ist das Klassendiagramm vom Transformationsprogramm. Zuerst ist die Klasse Excutable die „main class". Sie ruft die anderen Klassen an. Die Klasse CompositionGrammar produziert die Grammatiken über alle einzelne Templates, die in der XML-Datei vorkommen, und speichert sie in Datenaufbau „List< CompositionGrammar>". Die Klassen über einzelne Templates produzieren die Grammatiken entsprechendes Templates, z.B. die Klassen Existence, Existence2 usw. Es gibt 34 Klassen über einzelne Templates, deshalb werden nur einige Beispiele „Existence", „Response" und „ExclusiveChoice" in Abbildung 14 aufgeführt. Die Klasse MergingInits verschmilzt die Grammatiken von mehreren Templates *init()* und *strong init()*. Die Klasse MergingAlgorithmus verschmilzt zwei allgemeine Grammatiken zu einer Kompositionsgrammatik, z.B. eine neue Kompositionsgrammatik= MergingAlgorithmus(*response()*, *precedence()*). Die Klasse Rename behandelt die Indexe der Non-Terminals in den Grammatiken. Sie wandelt die komplexen Indexe zu einfachen Indexen um, z.B. $S_{(1,1)}$ zu $S_{(1)}$. Die Aufgabe der Klasse AnalyseNonTerminal ist, die Indexe und Symbole von Non-Terminals herauszuziehen. Z.B. kann die Klasse AnalyseNonTerminal $S_{(1,1)}$ einlesen, und gibt die Informationen zurück, dass S das Symbol ist und $_{(1,1)}$ der Index ist. Die Klasse GrammarOutput kann eine XML-Datei über die resultierende Grammatik mit vorgegebenen XML Schema herstellen. Die Klassen ServiceNonTerminal, ServiceParam und ServiceNonTerminalType sind selbst definierten Datenaufbaue. ServiceNonTerminal wird benutzt, die Non-Terminals mit Web Services zu speichern. Die Inhalte der Ein/Ausgabe-Parameters sind im Format ServiceParam gespeichert. Die Typen von Web Services werden im Format ServiceNonTerminalType definiert. Die Klassen NonTerminal, Terminal und Rule sind auch Datenaufbaue. Alle vorkommenden Non-Terminals in Grammatiken sind im Format NonTerminal definiert. Ähnlich ist die Klasse Terminal für alle Terminals in Grammatiken, während alle Produktionsregeln im Format Rule gespeichert werden.

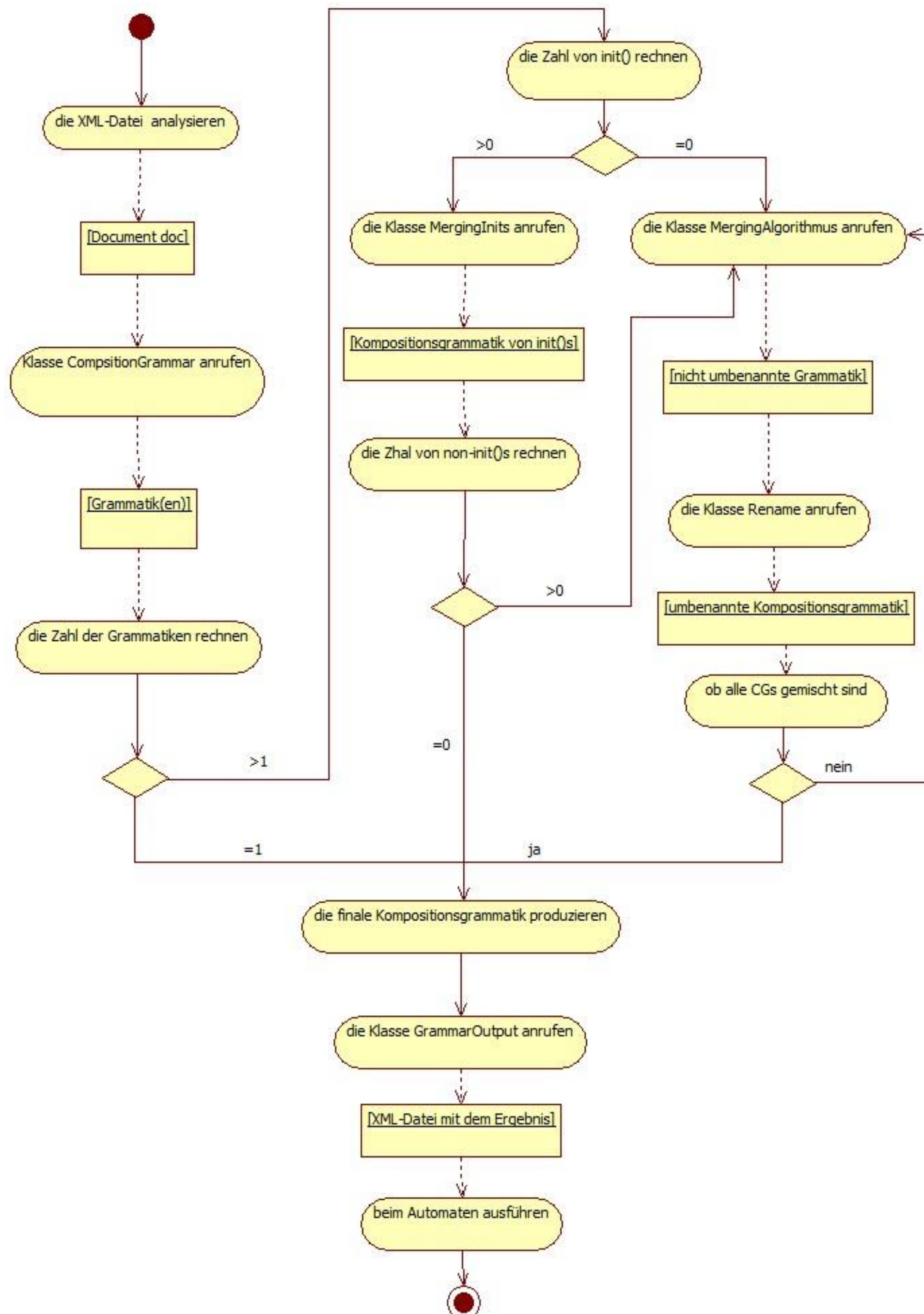
Das Programm beginnt mit der Klasse Excutable und bekommt eine Variable *Document doc*. Dann wird die Klasse CompositionGrammar angerufen, um die Variable *doc* zu behandeln. Alle Grammatiken von einzelnen Templates in der XML-Datei werden bekommen. Danach werden die Klassen MergingInits und MergingAlgorithmus angerufen, um die Kompositionsgrammatiken zu produzieren. Zum Schluss wird die Klasse GrammarOutput angerufen, um eine XML-Datei mit dem vorgegebenen Schema zu produzieren.



Die Abbildung 14. das Klassendiagramm vom Programm.

Die Abbildung 15 ist das Aktivitätsdiagramm vom Transformationsprogramm. Das Programm beginnt mit der Klasse „Excutable“. Die Grammatiken stammen aus der XML-Datei, deshalb das Programm die XML-Datei in der Klasse Excutable zuerst „parse“ muss. DOM von Java wird angewandt, um XML-Datei zu parse, wie der folgende Code.

```
Document doc = null;
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
factory.setNamespaceAware(true);
factory.setIgnoringElementContentWhitespace(true);
DocumentBuilder builder = factory.newDocumentBuilder();
doc = builder.parse("test/XXX.xml");
```



Die Abbildung 15. Das Aktivitätsdiagramm vom Programm.

Alle getesteten Dateien befinden im Pfad „/test/“. Die analysierte XML-Datei wird in der Variable „doc“ von Typ „Document“ gespeichert. Zuerst wird die Klasse „CompositionGrammar“ in der Klasse Excutable angerufen. Die Eingabe der Klasse CompositionGrammar ist die Variable doc, und seine Ausgabe ist eine oder mehrere produzierten Grammatiken von den einzelnen Templates, die in der XML-Datei vorkommen. Die Klasse CompositionGrammar produziert diese Grammatiken durch die entsprechenden Klassen einzelnes Templates. Wie die Abbildung 13 zeigt, jedes vorkommenden einzelnen Template wird in <constraint>...</constraint> einbezogen. Darin gibt es ein Element für das entsprechende Template, z.B. <name>existence</name>, das den Typ dieses Templates zeigt. Dadurch entscheidet das Programm, welche Grammatik zu produzieren ist. Dann ruft das Programm die entsprechende Templates Klasse an. Z.B. falls es <name>existence</name> ist, dann ruft das Programm die Klasse „Existence“ auf. Alle produzierten Grammatiken von einzelnen Templates werden im „List<CompositionGrammar> all_CG“ gespeichert. Dann entscheidet das Transformationsprogramm die Zahl der produzierten Grammatiken. Falls nur eine Grammatik hergestellt wird (Das bedeutet, in der XML-Datei gibt es nur ein Template.), ist sie die finale Grammatik. Falls mehrere Grammatiken produziert werden (Das bedeutet, die XML-Datei umfasst mehrere Templates), müssen die Grammatiken gemischt werden („*merge*“). Die Mischung von mehreren *init()* und *strong init()* ist ein Typ Operation von „∪“ und die Grammatiken sind kontext-sensitiv, während die Mischung der anderen allgemeinen Grammatiken ein Typ Operation von „∩“ ist und die Grammatiken kontext-frei sind[1]. Deshalb muss das Transformationsprogramm noch kontrollieren, ob Grammatiken von *init()* und *strong init()* in den produzierten Grammatiken (d.h. „List<CompositionGrammar> all_CG“) existieren. Falls es kein *init()* oder *strong init()* gibt (Das bedeutet, alle Grammtiken sind allgemeine Grammatiken, wie *response(), precedence()* usw.), wird die Klasse MergingAlgorithmus direkt wiederholt angerufen, bis alle Grammatiken in all_CG behandelt werden. Sonst (Es gibt *init()* oder *strong init()* in all_CG.) wird die Klasse MergingInits zuerst angerufen. Durch diese Klasse MergingInits wird die Kompositonsgrammatik aller in all_CG vorkommenden *init()* und *strong init()* produziert. Danach wird die Klasse MergingAlgorithmus für die Kompositionsgrammatik von MergingInits und die alle anderen allgemeinen Grammatiken wiederholt angewandt, bis alle Grammatiken in all_CG behandelt werden. Falls alle Grammatiken in all_CG schon behandelt werden, bekommen wir die finale Kompositionsgrammatik. Die finale Kompositionsgrammatik wird in „*CompositionGrammar result_Grammar*“ gespeichert. Bitte beachten, dass die Klasse MergingAlgorithmus zwei Grammatiken zu einer Grammatik mischen kann. Aber MergingAlgorithmus kann nur die „kleine“ Grammatik zur „großen“ Grammatik mischen[1]. Z.B. wird MergingAlgorithmus angerufen, um zwei Grammatiken G1 und G2 zu mischen, und wir bekommen die Grammatik G als Ergebnis. Dann mischen wir die dritte Grammatik G3 weiter. Wir können nur G3 in G mischen, aber dürfen nicht die Grammatik G in G3 mischen. Der Algorithmus ist links-assoziativ.

In der Prozedur wird die Klasse Rename angerufen, so dass die komplexen Indexe (z.B. $S_{(1,1)}$) zu den einfachen Indexen (z.B. $S_{(1)}$) umgewandelt werden. Zuerst wird die Klasse AnalyseNonTerminal von der Klasse Rename angerufen, um die Indexe von Non-Terminals herauszuziehen. Weil die produzierten einfachen Indexe sukzessiv sein sollen[1], behandelt die Klasse Rename zuerst die Indexe im Format (i,i) mit i=i. Die Klasse wandelt $S_{(i,i)}$ zu $S_{(i)}$ um. Der neue einfache Index i ist ein „i“ von (i,i), z.B. $S_{(1,1)}$ zu $S_{(1)}$, $S_{(2,2)}$ zu $S_{(2)}$. Die Zahl der Indexe im Format (i,i) ist n. Dann behandelt die Klasse Rename die Indexe (i,j) mit $i \neq j$. Sie wandelt $S_{(i,j)}$ zu $S_{(k)}$ um. Der neue einfache Index k beginnt mit n+1. Nach jeder Transformation wird „n=n+1“ angewandt, z.B. $S_{(1,2)}$ zu $S_{(3)}$, $S_{(2,3)}$ zu $S_{(4)}$. Deshalb sind die neuen einfachen Indexe sukzessiv.

Die andere Aufgabe der Klasse CompositionGrammar ist, die Ein/Ausgabe-Parameters und Web-Services-Informationen von Aktivitäten in XML-Datei auszugeben. Diese Ein/Ausgabe-Parameters werden als Attribute der Tags <dataelement id="X" initial="XX" name="XX" type="XX"> in XML-Datei gespeichert. Das Transformationsprogramm verwendet „doc.getElementsByTagName()" und „.getAttribute()", um die Attribute „id", „initial", „name" und „type" zu bekommen, wie die Abbildung 11 und 12 oben gezeigt haben. Das Programm baut ein „List<ServiceParam> inputoutput", und speichert alle Ein/Ausgabe-Parameters mit den bekommenen Informationen darin. Alle in ConDec Modell vorkommenden Aktivitäten werden in <activitydefinitions>...</activitydefinitions> einbezogen. Das Programm muss die Aktivitäten behandeln, deren Typen „Web Services" sind. Das bedeutet, dass diese Aktivitäten das Tag <remote task="XXX"> haben, wie die Abbildung 11 zeigt. Dann werden „.getElementsByTagName()" und „.getAttribute()" verwendet, wie oben erwähnt, um die Informationen von Web Services zu bekommen. Zum Schluss werden die Attribute „id", Typ, Eingabe und Ausgabe im Format ServiceNonTerminal in ein „Set<ServiceNonTerminal> servicenonterminal" gespeichert.

Nach der Produktion der finalen Kompositionsgrammatik wird die Klasse „GrammarOutput" in der Klasse Excutable angerufen. Die Eingaben von GrammarOutput sind das *Set<ServiceNonTerminal> servicenonterminal*, d.h. alle Non-Terminals mit dem Typ von Web Services, und *CompositionGrammar result_Grammar*, d.h. die finale Kompositionsgrammatik. Die Ausgabe ist eine XML-Datei, die „compositiongrammar.xml" heißt. Die Datei „compositiongrammar.xml" befindet sich im Pfad „\lsg\grammarfile". Die Non-Terminals, Typen von Non-Terminals, Ein/Ausgabe-Parameters, Terminals, Produktionsregeln, Startsymbol usw. werden in dieser XML-Datei mit vorgegebenen XML Schema gespeichert.

3.2 Herausforderungen

Das Transformationsprogramm ist auf das declare-2.2.0 basierend. Die Software kann den ConDec Modell strukturieren, aber es gibt noch einige Fragen, die es im Detail zu behandeln gilt.

3.2.1 Existence Templates

Strong init(A) und init(A): In declare-2.2.0 werden die Existence Templates *strong init(A)* und *init(A)* angeboten. Die Arbeiten[1, 2, 16, 17] betreffen den Unterschied von den beiden Restriktionen nicht. Jetzt diskutieren wir ihren Unterschied. Das LTL von *init(A)* in der durch declare-2.2.0 produzierten XML-Datei ist :**(("A.started" \/ "A.cancelled") W "A").** Das LTL von *strong init(A)* ist: **(("A.started" \/ "A.cancelled") U "A.completed")**. Der Unterschied ist nur „weak until A" und „until A.completed". Bei „weak until A" ist A nicht bestimmt wahr. Bei „until A.completed" muss A „completed" sein. Aber die Kompositionsgrammatik behandelt nur die Beziehungen zwischen Aktivitäten[1]. Es wird nicht berücksichtigt, ob A „started" oder „completed" ist. Deshalb für die Kompositionsgrammatik sind die beiden LTL-Formeln gleich. Ihre Grammatiken sind auch gleich, wie die Abbildung 16.

$S_0 \rightarrow A_0\, H_0$ with: $A_i, B_i, C_i \in$ Services
$A_0 \rightarrow a$ $S_i, H_i \in$ Helpers
$a\, H_0 \rightarrow a\, S_1$ $a, b, c \in \Sigma$
$S_1 \rightarrow A_1 | B_1 | C_1 | \varepsilon$
$A_1 \rightarrow aS_1$
$B_1 \rightarrow bS_1$
$C_1 \rightarrow cS_1$

Die Abbildung 16. Die Produktionsregeln von *strong init(A) und init(A)* sind gleich.

Last(A). Die Arbeiten [1, 2] erwähnt das nicht. In declare-2.2.0 gibt es das Template *last(A)*, aber keine Beschreibung. In der durch declare-2.2.0 produzierten XML-Datei ist das LTL von *last(A)*: **[] ("A" -> !X!"A")**. Das bedeutet, dass A die letzte Aktivität im Modell sein muss. Die Grammatik wurde von mir geschrieben, wie die Abbildung 17 zeigt. In einigen Arbeiten wird der Operator „next“ durch „○“ präsentiert. Aber in den durch declare-2.2.0 produzierten XML-Dateien repräsentiert „**X**“ den Operator „next“.

$S_1 \rightarrow A_1 | B_1 | C_1 | \varepsilon$ with: $A_i, B_i, C_i \in$ Services
$A_1 \rightarrow aS_2$ $S_i \in$ Helpers
$B_1 \rightarrow bS_1$ $a, b, c \in \Sigma$
$C_1 \rightarrow cS_1$
$S_2 \rightarrow A_2 | \varepsilon$
$A_2 \rightarrow aS_2$

Die Abbildung 17. Die Produktionsregeln von *last(A).* A ist die letzte Aktivität.

Error(A). In declare-2.2.0 hat das Template *error(A)* keine Beschreibung. Das LTL von *error(A)* ist: **(<> ("A.completed") /\ !(<> ("A.started"))).** Das bedeutet, dass A schließlich nicht startet aber fertig ist. Das LTL von *absence(A)* ist: **!(<> ("A.started")).** D.h. A startet gar nicht. Der Unterschied zwischen *error(A)* und *absence(A)* ist der Zustand „**<>("A.completed")**“. Bedingt durch die Kompositionsgrammatik, die keine Zustände von Aktivitäten behandelt, nehme ich an, dass die Produktionsregeln von *error(A)* und *absence(A)* gleich sind. Die Abbildung 18 zeigt die Produktionsregeln.

$S_1 \rightarrow B_1 | C_1 | \varepsilon$ with: $B_i, C_i \in$ Services
$B_1 \rightarrow bS_1$ $S_i \in$ Helpers
$C_1 \rightarrow cS_1$ $a, b, c \in \Sigma$

Die Abbildung 18. Die Produktionsregeln von *error(A)* und *absence(A)* sind gleich.

3.2.2 Relation Templates

Alternate(A,B) und alternate response(A,B). In der Arbeit[1] gibt es *alternate response(A,B)*, aber keine *alternate(A,B).* Die beide kommen in declare-2.2.0 vor. Das LTL-Formel von *alternate(A,B)* ist:**[](("A" -> X((!("A") W "B"))))**. Seine Beschreibung in declare-2.2.0 ist: „If A is excuted, then next A can not be excuted before B is excuted after the previous A.“

Z.B. die Folgen CB und ACBACBBAC befriedigen diese Restriktion, die Folge ACAB nicht. Das LTL von *alternate response(A,B)* ist:**[](("A" -> X((!("A") U "B")))).** Die Beschreibung in declare-2.2.0 ist: „After each A is excuted at least one B is excuted. Another A can be excuted again only after the first B.“ Der Unterschied ist „weak until B“ und „until B“. Für *alternate(A,B)* kann B nicht geschehen. Für *alternate response(A,B)* muss B nach jedem A mindestens einmal geschehen. Deshalb nehme ich an, dass die beiden Produktionsregeln gleich sind, wie die Abbildung 19 zeigt.

$S_1 \rightarrow A_1 | B_1 | C_1 | \varepsilon$ with: $A_i, B_i, C_i \in$ Services
$A_1 \rightarrow aS_2$ $S_i \in$ Helpers
$B_1 \rightarrow bS_1$ $a, b, c \in \Sigma$
$C_1 \rightarrow cS_1$
$S_2 \rightarrow B_2 | C_2$
$B_2 \rightarrow bS_1$
$C_2 \rightarrow cS_2$

Die Abbildung 19. Die Produktionsregeln von *alternate(A,B)* und *alternate response(A,B)* sind gleich.

3.2.3 Choice Templates

choice(A,B). In den Arbeiten[1, 2] wird das Template *choice(A,B)* nicht diskutiert. Aber in declare-2.2.0 kommt es vor. Das LTL von *choice(A,B)* ist:**(<> ("A") \/ <>("B")).** Die Beschreibung ist „At least one from A and B has to be excuted“. Seine LTL und Beschreibung sind ähnlich wie das Template *1 of 2(A,B)*. Das LTL von *1 of 2(A,B)* ist: **<>(("A" \/ "B"))**. Die Beschreibung in declare-2.2.0 ist „Either A is excuted at least once, or B is excuted at least once“. Ich glaube, dass die beiden LTL gleich sind. Deshalb sind ihre Produktionsregeln auch gleich, wie die Abbildung 20. Der Unterschied in declare-2.2.0 ist, dass *1 of 2(A,B)* „not branchable“ ist und *choice(A,B)* „branchable“ sein kann. Z.B. ist *choice*($(A_1,A_2), (B_1,B_2,B_3)$) in declare-2.2.0 gültig. Aber *1 of 2*($(A_1,A_2), (B_1,B_2,B_3)$) ist in declare-2.2.0 nicht erlaubt. Ich glaube, die Produktionsregeln von *choice 1 of 3(A,B,D)* im declare-2.2.0 und *1 of 3(A,B,D)* in der Arbeit[2] sind auch gleich. Das LTL von *choice 1 of 3(A,B,D)* in declare-2.2.0 ist:**((<>("A") \/ <>("B")) \/ <>("C")).** Das LTL von *1 of 3(A,B,D)* in der Arbeit[2] ist **<>("A") \/ <>("B")) \/ <>("C").** Deshalb kann ich vermuten, dass diese beiden Restriktionen gleich sind.

$S_1 \rightarrow A_1 | B_1 | C_1$ with: $A_i, B_i, C_i \in$ Services
$A_1 \rightarrow aS_2$ $S_i \in$ Helpers
$B_1 \rightarrow bS_2$ $a, b, c \in \Sigma$
$C_1 \rightarrow cS_1$
$S_2 \rightarrow A_2 | B_2 | C_2 | \varepsilon$
$A_2 \rightarrow aS_2$
$B_2 \rightarrow bS_2$
$C_2 \rightarrow cS_2$

Die Abbildung 20. Die Produktionsregeln von *1 of 2(A,B)* und *choice(A,B)* sind gleich.

Exclusive choice 1 of 3(A,B,D). In der Arbeit[1] gibt es kein *Exclusive choice 1 of 3(A,B,D)*. Seine Beschreibung in declare 2.2.0 ist „At least two activities from (A,B,C) have to be excuted.“ Ich glaube, diese Beschreibung ist falsch. Sein LTL in declare-2.2.0 ist:**(((((<>("A") \/ <>("B")) \/ <>("C")) /\ !((<>("A") /\ <>("B")))) /\ !((**

<>("B") /\ <>("C")))) /\ !((<>("A") /\ <>("C")))). Das bedeutet, dass nur eine Aktivität ausgeführt ist, während die anderen zwei Aktivitäten nicht ausgeführt sein dürfen. Darüber hinaus ist das LTL von *exclusive 1 of 3(A,B,D)* in der Arbeit[2] auch falsch. Die Beschreibung in der Arbeit[2] ist „One of the events A,B or C has to eventually occur, but the other two can not occur at all.“ Aber sein LTL ist: **(<>("A") /\ !<>("B")) /\ !<>("C")) \/(!<>("A") /\ <>("B")) /\ !<>("C"))\/(<>("A") /\ !<>("B")) /\ !<>("C"))**. **<>("A") /\ !<>("B")) /\ !<>("C")** bedeutet, A ist ausgeführt aber B und C nicht ausgeführt sind. **!<>("A") /\ <>("B")) /\ !<>("C")**bedeutet, B ist ausgeführt aber A und C nicht. Deutlich ist der rote Teil falsch. Dieser Teil soll „C ist ausgeführt aber A und B dürfen nicht“ sein. Das richtige LTL vom roten Teil ist: **!<>("A") /\ !<>("B")) /\ <>("C")**. Das ganze LTL von *exclusive 1 of 3(A,B,C)* ist: **(<>("A") /\ !<>("B")) /\ !<>("C")) \/(!<>("A") /\ <>("B")) /\ !<>("C")) \/(!<>("A") /\ !<>("B")) /\ <>("C"))**. Dieses LTL ist gleich wie das LTL von *exclusive choice 1 of 3(A,B,D)* im declare-2.2.0. Deshalb glaube ich, dass die Restriktionen *exclusive choice 1 of 3(A,B,D)* und *exclusive 1 of 3(A,B,D)* equivalent sind und die Beschreibung von *exclusive choice 1 of 3(A,B,D)* im declare-2.2.0 falsch ist. Seine richtige Beschreibung soll „only one activity is excuted, the other two activities must not be excuted“ sein.

3.2.4 Branching von Restriktionen

Im declare-2.2.0 dürfen einige Restriktionen nicht „branchable“ sein, z.B. *chain response(A,B)*, *choice 1 of 3(A,B,D)*, *choice 2 of 3(A,B,D)*... usw. Eine XML-Datei mit der „branching“ Situation von diesen Restriktionen wird von declare-2.2.0 nicht geöffnet. Z.B. kann declare 2.2.0 die XML-Datei mit *choice 1 of 3((*A_1*,*A_2*),B,D)* nicht öffnen. Kein Modell wird in declare 2.2.0 vorgezeigt. Aber theoretisch sind diese Restriktionen in ConDec „branchable“[1, 2]. Deshalb sollen wir die „branching“ Situationen von diesen Restriktionen berücksichtigen. Das Programm kann die „branching“ Situationen behandeln.

3.3 Testfälle

Das Programm muss getestet werden. In dem Abschnitt werden zwei geprüfte Beispiele vorgestellt, das Beispiel „religion“ und das Beispiel „medical“. Sie sind die eigenen Beispiele von declare 2.2.0. Zuerst sehen wir das Beispiel „religion“. Es befindet sich im Pfad „\declare-2.2.0\examples\religion\religion.xml“. Die Abbildung 9 ist die graphische Repräsentation vom ConDec Modell „religion“. Nach der Analyse von der XML-Datei produziert das Transformationsprogramm drei Grammatiken von einzelnen Templates: *existence(pray)*, *not co-existence(curse, become holy)* und *response(curse, pray)*. Ihre ausführlichen Produktionsregeln sind einfach und werden hier nicht beschrieben. Z.B. G1 ist die Grammatik von *existence(pray),* G2 ist die Grammatik von *not co-existence(curse, become holy)* und G3 ist die Grammatik von *response(curse, pray).* Dann wird MergingAlgorithmus(G1,G2) angewendet und die Kompositionsgrammatik CG bekommen, d.h. die Kompositionsgrammatik von *existence(pray)* und *not co-existence(curse, become holy).* Die Abbildung 21 zeigt diese Kompositionsgrammatik CG. Die Klasse MergingAlgorithmus beginnt mit S(1,1). In der Grammatik von *existence(pray)* gibt es S(1)→curse(1)|pray(1)|bless(1)|become.holy(1). In der Grammatik von *not co-existence(curse, become holy)* gibt es S(1)→curse(1)|pray(1)|bless(1)|become.holy(1)|ε. Nach dem Merging-Algorithmus von allgemeinen Grammatiken[1] kann S(1,1)→ curse(1,1)|pray(1,1)|bless(1,1)|become.holy(1,1) bekommen werden. S(1)→ε gehört nicht zu *existence(pray)* (G1), aber nur zu *not co-existence(curse, become holy)* (G2), deshalb es gelöscht wird. In G1 existiert curse(1)→curse_t S(1). In G2 existiert curse(1)→curse_t S(2).

Deshalb wird curse(1,1)→curse_t S(1,2) in CG hinzugefügt. S(1,2) wird als neues unbearbeitetes Non-Terminal in die Menge aller unbearbeiteten S(u,v) hinzugefügt. Diese Menge heißt ***S***. Alle behandelten S(u,v) werden aus ***S*** gelöscht. Ähnlich wird pray(1,1)→pray_t S(2,1) in CG hinzugefügt und S(2,1) in die Menge ***S*** hinzugefügt. Durch die gleiche Methode wird bless(1,1)→bless_t S(1,1) in CG hinzugefügt. Aber S(1,1) wird schon bearbeitet, deshalb wird S(1,1) nicht in die Menge ***S*** hinzugefügt. Die Klasse wiederholt diese Schritte, bis alle Non-Terminals in ***S*** bearbeitet werden. Dann bekommen wir die Kompositionsgrammatik von *existence(pray)* und *not co-existence(curse, become holy)*, wie die Abbildung 21 zeigt. Durch die Klasse Rename werden die komplexen Indexe zu einfache Indexe umgewandelt. In der Abbildung 21 gibt es (1,1), (2,2), (1,2), (2,1), (1,3) und (2,3). Zuerst wandelt die Klasse Rename (1,1) zu (1), (2,2) zu (2) um, wie es im Abschnitt 3.1 vorgestellt wird. Die restlichen Indexe sollen mit „3“ beginnen. Bitte beachten Sie, dass die Non-Terminals im Datenaufbau „Set< INonterminal >“ gespeichert werden. Die Reihenfolge von (1,2), (2,1), (1,3), (2,3) ist nicht bestimmt. Falls die Reihenfolge der Behandlung (1,2), (2,1), (1,3), (2,3) ist, werden (1,2) zu (3), (2,1) zu (4), (1,3) zu (5) und (2,3) zu (6) umgewandelt. Falls die Reihenfolge der Behandlung (2,3), (1,2), (2,1), (1,3) ist, werden (2,3) zu (3), (1,2) zu (4), (2,1) zu (5) und (1,3) zu (6) umgewandelt. Vielleicht sind die neuen Indexe zweimal unterschiedlich, aber die beiden Ergebnisse sind richtig. (1,1) zu (1), (2,2) zu (2), (3) bis (6) werden zu (1,2), (2,1), (1,3), (2,3) zufällig zugewiesen.

```
S(1,1)  --> curse(1,1)|pray(1,1)|bless(1,1)| become.holy(1,1)
curse(1,1)  --> curse_t S(1,2)
pray(1,1)  --> pray_t S(2,1)
bless(1,1)  --> bless_t S(1,1)
become.holy(1,1)  --> become.holy_t S(1,3)
S(2,2)  --> curse(2,2)| pray(2,2)| bless(2,2)|ε
curse(2,2)  --> curse_t S(2,2)
pray(2,2)  --> pray_t S(2,2)
bless(2,2)  --> bless_t S(2,2)
S(2,3)  --> pray(2,3)| bless(2,3)| become.holy(2,3)|ε
pray(2,3)  --> pray_t S(2,3)
bless(2,3)  --> bless_t S(2,3)
become.holy(2,3)  --> become.holy_t S(2,3)
S(1,3)  --> pray(1,3)| bless(1,3)| become.holy(1,3)
pray(1,3)  --> pray_t S(2,3)
bless(1,3)  --> bless_t S(1,3)
become.holy(1,3)  --> become.holy_t S(1,3)
S(1,2)  --> curse(1,2)| pray(1,2)|bless(1,2)
curse(1,2)  --> curse_t S(1,2)
pray(1,2)  --> pray_t S(2,2)
bless(1,2)  --> bless_t S(1,2)
S(2,1)  --> curse(2,1)| pray(2,1)| bless(2,1)| become.holy(2,1)|ε
curse(2,1)  --> curse_t S(2,2)
pray(2,1)  --> pray_t S(2,1)
bless(2,1)  --> bless_t S(2,1)
become.holy(2,1)  --> become.holy_t S(2,3)
```

Die Abbildung 21. Die Kompositionsgrammatik von *existence(pray)* und *not co-existence(curse, become holy).*

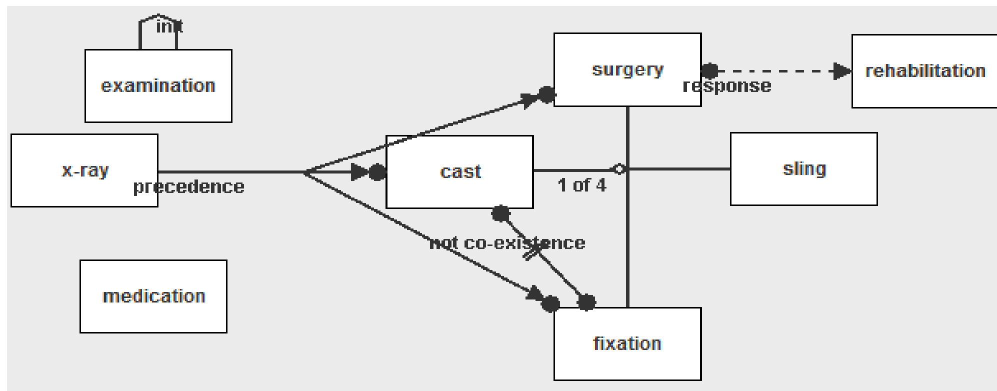
Dann werden alle komplexen Indexe der Kompositionsgrammatik CG zu einfachen Indexe umgewandelt. In diesem Beispiel sind (1,1)→(1), (2,2)→(2), (1,2)→(3), (2,1)→(4), (1,3)→(5) und (2,3)→(6). Danach wird MergingAlgorithmus(CG,G3) wieder benutzt, d.h. das Programm verschmilzt G3 zur erhaltenen Grammatik CG. Die produzierte Grammatik ist auf Abbildung 22 zu sehen. Dann wird die Klasse Rename nochmal benutzt. Zum Schluss bekommen wir die finale Kompositionsgrammatik, die sich im Anhang befindet.

```
S(1,1)  --> curse(1,1)| pray(1,1)| bless(1,1)|become.holy(1,1)
curse(1,1)  --> curse_t S(3,2)
pray(1,1)  --> pray_t S(4,1)
bless(1,1)  --> bless_t S(1,1)
become.holy(1,1)  --> become.holy_t  S(5,1)
S(2,2)  --> curse(2,2)|pray(2,2)|bless(2,2)
curse(2,2)  --> curse_t S(2,2)
pray(2,2)  --> pray_t S(2,1)
bless(2,2)  --> bless_t S(2,2)
S(6,1)  --> pray(6,1)| bless(6,1)|become.holy(6,1)|ε
pray(6,1)  --> pray_t S(6,1)
bless(6,1)  --> bless_t S(6,1)
become.holy(6,1)  --> become.holy_t  S(6,1)
S(2,1)  --> curse(2,1)| pray(2,1)| bless(2,1)|ε
curse(2,1)  --> curse_t S(2,2)
pray(2,1)  --> pray_t S(2,1)
bless(2,1)  --> bless_t S(2,1)
S(4,1)  --> curse(4,1)| pray(4,1)| bless(4,1)| become.holy(4,1)|ε
curse(4,1)  --> curse_t S(2,2)
pray(4,1)  --> pray_t S(4,1)
bless(4,1)  --> bless_t S(4,1)
become.holy(4,1)  --> become.holy_t  S(6,1)
S(5,1)  --> pray(5,1)| bless(5,1)|become.holy(5,1)
pray(5,1)  --> pray_t S(6,1)
bless(5,1)  --> bless_t S(5,1)
become.holy(5,1)  --> become.holy_t  S(5,1)
S(3,2)  --> curse(3,2)| pray(3,2)| bless(3,2)|
curse(3,2)  --> curse_t S(3,2)
pray(3,2)  --> pray_t S(2,1)
bless(3,2)  --> bless_t S(3,2)
```

Die Abbildung 22. Die Kompositionsgrammatik von CG und *response(curse, pray)*.

Jetzt testen wir ein anderes Beispiel „medical“. Es befindet sich im Pfad „\declare-2.2.0\declare-2.2.0\examples\medical.xml“. Die Abbildung 23 ist das ConDec Modell vom Beispiel „medical“. Zuerst bekommen wir die sechs Grammatiken: *init(examination), init(examination), precedence(x-ray,(surgery,cast,fixation)), choice 1 of 4(cast,surgery,sling,fixation), response(surgery rehabilitation)* und *not co-existence(cast, fixation).* Diese sechs Grammatiken heißen G1, G2, G3, G4, G5 und G6. Weil das Template

init() existiert, wird die Klasse MergingInits zuerst angerufen. Diese Klasse mischt alle *init()* (G1 und G2 in diesem Beispiel) zusammen. Nach dem „Join-Algorithmus“[1] bekommen wir die Zwischengrammatik CG als die Kompositionsgrammatik von mehreren *init()*. Weil G1 und G2 gleich sind, ist das Ergebnis der Komposition auch *init(examination)*. Die resultierende Grammatik beginnt mit S(0). Jetzt wird MergingAlgorithmus(CG,G3) angewendet, d.h. MergingAlgorithmus(Kompositionsgrammatik mehrerer *init()*, *precedence(x-ray,(surgery,cast,fixation))*). Dann wird die Klasse Rename benutzt und eine neue Zwischengrammatik bekommen. Das Ergebnis ist wie die Abbildung 24. Die oben erwähnten Schritte werden wiederholt, bis alle Grammatiken behandelt werden. Zum Schluss bekommen wir die finale Kompositionsgrammatik, die sich im Anhang befindet.



Die Abbildung 23. Das Beispiel „medical“ im declare-2.2.0.

```
S(0)  --> examination(0) H(0)
examination(0)  --> examination_t
examination_t H(0)  --> S(1)
S(1)  --> examination(1) | medication(1) | rehabilitation(1) | sling(1) |
             x-ray(1)|ε
examination(1)  --> examination_t S(1)
medication(1)  --> medication_t S(1)
rehabilitation(1)  --> rehabilitation_t S(1)
sling(1)  --> sling_t S(1)
x-ray(1)  --> x-ray_t S(2)
S(2)  --> examination(2) | medication(2) | rehabilitation(2) | sling(2) |
            surgery(2) |x-ray(2) |cast(2) | fixation(2) | ε
examination(2)  --> examination_t S(2)
medication(2)  --> medication_t S(2)
rehabilitation(2)  --> rehabilitation_t S(2)
sling(2)  --> sling_t S(2)
surgery(2)  --> surgery_t S(2)
x-ray(2)  --> x-ray_t S(2)
cast(2)  --> cast_t S(2)
fixation(2)  --> fixation_t S(2)
```

Die Abbildung 24. Die Kompositionsgrammatik von MergingAlgorithmus(CG, *precedence(x-ray,(surgery,cast,fixation))*)

4 Zusammenfassung

In der Arbeit wird die Implementierung einer Transformation von der deklarativen Sprache ConDec zur Kompositionsgrammatiken diskutiert. In der Forschung besteht die Hoffnung, dass der assembler-ähnliche Charakter von Kompositionsgrammatiken die Grundlage für die unifizierte Modellierung bilden kann. Zuerst werden Web Services und ihre Kompositionen in der Arbeit vorgestellt. Web Services sind die Abstraktionen für verschiedene Plattformen und Sprachen, so dass die Aufrufe von Web Services gleich sind. Ihre Kompositionen beschreiben die Art und Weise wie Web Services miteinander verknüpft sind. Dann wird der Begriff deklarative Sprache beschrieben. Die Vorgehensweise von imperativen Sprachen ist „*say how to do something*“. Zum Unterschied von imperativen Sprachen ist die Vorgehensweise deklarativer Sprachen „*say what is required and let the system determine how to achieve it*“. Durch den Vergleich ist es klar, dass die deklarativen Sprachen die beiden Anforderungen zwischen Unterstützung und Flexibilität von Systemen gut ausgleichen können. Danach werden ConDec und die Software Declare vorgestellt. ConDec ist eine deklarative Sprache und die Software Declare unterstützt ConDec. Dann wird die Implementierung der Transformation ausführlich beschrieben. Durch das Klassendiagramm werden die Funktionen der Klassen vom Transformationspragramm vorgestellt. Das Aktivitätsdiagramm beschreibt den Ablauf vom Transformationspragramm. Darüber hinaus wird eine detaillierte Diskussion über die Herausforderungen der Transformation geführt. Die Diskussion ist über einige Templates, die in der Arbeit[1] nicht vorkommen. Zum Schluss werden zwei Testfälle geprüft, die die eigenen Beispiele in declare-2.2.0 sind.

Anhang

A.1 Die finale Kompositionsgrammatik vom Beispiel „religion“

S(1) --> pray(1)

pray(4) --> pray_t S(4)

bless(2) --> bless_t S(2)

curse(1) --> curse_t S(3)

curse(5) --> curse_t S(2)

S(5) --> curse(5)

bless(6) --> bless_t S(6)

S(5) -->ε

S(5) --> pray(5)

become.holy(7) --> become.holy_t S(7)

S(1) --> bless(1)

S(4) -->ε

curse(3) --> curse_t S(3)

bless(7) --> bless_t S(7)

S(5) --> become.holy(5)

S(6) --> bless(6)

bless(5) --> bless_t S(5)

become.holy(5) --> become.holy_t S(6)

pray(3) --> pray_t S(4)

become.holy(1) --> become.holy_t S(7)

S(2) --> pray(2)

bless(3) --> bless_t S(3)

S(3) --> bless(3)

S(3) --> curse(3)

S(2) --> curse(2)

pray(1) --> pray_t S(5)

S(4) --> pray(4)

S(7) --> bless(7)

S(1) --> become.holy(1)

become.holy(6) --> become.holy_t S(6)

S(6) -->ε

S(2) --> bless(2)

pray(2) --> pray_t S(4)

S(7) --> become.holy(7)

S(5) --> bless(5)

S(6) --> become.holy(6)

pray(5) --> pray_t S(5)

S(1) --> curse(1)

curse(4) --> curse_t S(2)

S(3) --> pray(3)

S(4) --> curse(4)

curse(2) --> curse_t S(2)

pray(6) --> pray_t S(6)

bless(1) --> bless_t S(1)

S(6) --> pray(6)

S(4) --> bless(4)

bless(4) --> bless_t S(4)

S(7) --> pray(7)

pray(7) --> pray_t S(6)

A.2 Die finale Kompositionsgrammatiken vom Beispiel „medical“.

S(7) --> examination(7)

S(9) --> sling(9)

S(7) --> medication(7)

S(6) --> surgery(6)

S(8) --> x-ray(8)

rehabilitation(4) --> rehabilitation_t S(4)

sling(8) --> sling_t S(8)

examination(9) --> examination_t S(9)

S(6) --> cast(6)

x-ray(1) --> x-ray_t S(9)

S(5) --> x-ray(5)

examination(0) --> examination_t

fixation(9) --> fixation_t S(3)

surgery(4) --> surgery_t S(2)

S(3) --> examination(3)

examination(7) --> examination_t S(7)

S(4) --> surgery(4)

S(7) --> x-ray(7)

S(8) --> rehabilitation(8)

S(3) --> rehabilitation(3)

S(6) --> sling(6)

S(9) --> rehabilitation(9)

S(6) --> medication(6)

examination(8) --> examination_t S(8)

medication(6) --> medication_t S(6)

S(0) --> examination(0) H(0)

S(3) --> ε

fixation(5) --> fixation_t S(5)

x-ray(3) --> x-ray_t S(3)

S(4) --> cast(4)

medication(1) --> medication_t S(1)

x-ray(8) --> x-ray_t S(6)

sling(9) --> sling_t S(6)

cast(9) --> cast_t S(4)

S(8) --> examination(8)

S(2) --> cast(2)

S(7) --> rehabilitation(7)

S(4) --> sling(4)

medication(8) --> medication_t S(8)

S(4) --> medication(4)

sling(5) --> sling_t S(5)

S(6) --> examination(6)

cast(4) --> cast_t S(4)

rehabilitation(9) --> rehabilitation_t S(9)

S(4) --> rehabilitation(4)

rehabilitation(7) --> rehabilitation_t S(6)

S(9) --> examination(9)

examination(5) --> examination_t S(5)

examination(1) --> examination_t S(1)

cast(7) --> cast_t S(2)

rehabilitation(5) --> rehabilitation_t S(3)

S(9) --> cast(9)

rehabilitation(6) --> rehabilitation_t S(6)

S(1) --> medication(1)

sling(1) --> sling_t S(8)

S(7) --> fixation(7)

sling(7) --> sling_t S(7)

medication(5) --> medication_t S(5)

sling(2) --> sling_t S(2)

S(5) --> rehabilitation(5)

S(7) --> sling(7)

S(2) --> medication(2)

S(1) --> examination(1)

S(8) --> ε

S(1) --> sling(1)

S(9) --> x-ray(9)

examination(4) --> examination_t S(4)

x-ray(2) --> x-ray_t S(2)

x-ray(9) --> x-ray_t S(9)

S(6) --> x-ray(6)

S(7) --> cast(7)

surgery(7) --> surgery_t S(7)

examination(6) --> examination_t S(6)

surgery(6) --> surgery_t S(7)

rehabilitation(8) --> rehabilitation_t S(8)

x-ray(5) --> x-ray_t S(5)

S(1) --> rehabilitation(1)

fixation(3) --> fixation_t S(3)

S(5) --> surgery(5)

rehabilitation(1) --> rehabilitation_t S(1)

surgery(9) --> surgery_t S(7)

examination(2) --> examination_t S(2)

S(7) --> surgery(7)

medication(9) --> medication_t S(9)

S(6) --> ε

surgery(2) --> surgery_t S(2)

S(4) --> x-ray(4)

S(5) --> fixation(5)

sling(3) --> sling_t S(3)

S(3) --> surgery(3)

fixation(6) --> fixation_t S(3)

rehabilitation(2) --> rehabilitation_t S(4)

medication(7) --> medication_t S(7)

S(9) --> surgery(9)

S(2) --> examination(2)

S(4) --> ε

examination_t H(0) --> examination_t S(1)

x-ray(4) --> x-ray_t S(4)

cast(6) --> cast_t S(4)

x-ray(6) --> x-ray_t S(6)

S(9) --> medication(9)

fixation(7) --> fixation_t S(5)

S(3) --> fixation(3)

medication(3) --> medication_t S(3)

S(5) --> medication(5)

S(6) --> fixation(6)

S(4) --> examination(4)

cast(2) --> cast_t S(2)

S(2) --> surgery(2)

S(2) --> rehabilitation(2)

sling(6) --> sling_t S(6)

S(9) --> fixation(9)

S(8) --> sling(8)

S(3) --> sling(3)

S(6) --> rehabilitation(6)

S(8) --> medication(8)

rehabilitation(3) --> rehabilitation_t S(3)

examination(3) --> examination_t S(3)

x-ray(7) --> x-ray_t S(7)

surgery(3) --> surgery_t S(5)

S(5) --> examination(5)

medication(2) --> medication_t S(2)

S(3) --> medication(3)

medication(4) --> medication_t S(4)

surgery(5) --> surgery_t S(5)

sling(4) --> sling_t S(4)

S(2) --> sling(2)

S(3) --> x-ray(3)

S(1) --> x-ray(1)

S(2) --> x-ray(2)

S(5) --> sling(5)

Literaturverzeichnis

[1] Görlach. Katharina: *A Generic Transformation of Existing Service Composition Models to a Unified Model, Fakultät Informatik, Elektrotechnik und Informationstechnik*, Universität Stuttgart, 2013

[2] Maja, Pesic: *Constraint-Based Workflow Management Systems: Shifting Control to Users*,Technische Universität Eindhoven, 2008

[3] Biplav Srivastava, Jana Koehler: *Web Service Composition - Current Solutions and Open Problems*, IBM India Research Laboratory & IBM Zurich Research Laboratory, 2003

[4] T. Bellwood, u.a.: *Universal Description, Discovery and Integration specification (UDDI) 3.0*, http://uddi.org/pubs/uddi-v3.00-published-20020719.htm, 2002

[5] R. Chinnici, u.a.: *Web Services Description Language(WSDL) 1.1*, http://www.w3.org/TR/wsdl, 2001

[6] D. Box, u.a.: *Simple Object Access Protocol(SOAP) 1.2*, http://www.w3.org/TR/SOAP, 2007

[7] H. Foster, S.Uchitel, J. Magee, J. Kramer: *Model-based Verification of Web Service Compositions*, Department of Computing, Imperial College London, 2003.

[8] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, S. Weerawarana: *Business Process Execution Language For Web Services, Version 1.1*, 2003.

[9] D. Fensel, C. Bussler, Y. Ding, B.Omelayenko: *The Web Service Modeling Framework WSMF*, Vrije Universiteit Amsterdam & Oracle Corporation, 2002.

[10] D. Skogan, R. Gronmo, I. Solheim: *Web Service Composition in UML*, Enterprise Distributed Object Computing Conference, 2004.

[11] M.Jaeger, G. Rojec-Goldmann, G. Muehl: *QoS Aggregation for Web Service Composition using Workflow Patterns*, Technische Universität Berlin, 2004

[12] J. Rao, X. Su: *A Survey of Automated Web Service Composition Methods,* Department of Computer and Information Science, S.43-54, Springer-Verlag, 2005.

[13] P.V.Roy, S.Haridi: *Concepts, Techniques, and Models of Computer Programming*, Swedish Institute of Computer Science, 2004

[14] P. Pichler, B. Weber, S. Zugal, J. Pinggera, J. Mendling, H. Reijers: *Imperative versus Declarative Process Modelling Languages: An Empirical Inverstigation*, University of Innsbruck, Humboldt-Universität zu Berlin, Eindhoven University of Technology, 2011.

[15] van der Aalst, Jablonski S: *Dealing with workflow change: identification of issues and solutions*, International Journal of Computer Systems Science & Engineering, vol. 15 no.5, 2000

[16] W.M.P. van der Aalst, M.Pesic, H. Schonenberg: *Declarative workflows: Balancing between flexibility and support*, S.99-113, Springer-Verlag, 2009

[17] M. Pesic, W. M. P. van der Aalst: *A Declarative Approach for Flexible Business Processes Management*. In Proceedings of the BPM 2006 Workshops, volume 4103 of Lecture Notes in Computer Science, S.169-180, Springer-Verlag, 2006

[18] M.Weske, W.M.P.van der Aalst, H.M.W.Verbeek: *Advances in business process management*, Data & Knowledge Engineering 50, 2004

[19] W.M.P. van der Aalst, A.H.M. ter Hofstede, M.Weske: *Business Process Management*, Eindhoven University of Technology, Queensland University of Technology und University of Potsdam, S. 1-12, Springer-Verlag, 2003

Erklärung

Hiermit versichere ich, diese Arbeit selbstständig verfasst und nur die angegebenen Quellen benutzt zu haben. Wörtliche und sinngemäße Übernahmen aus anderen Quellen habe ich nach bestem Wissen und Gewissen als solche kenntlich gemacht.

Stuttgart, ______________________ ______________________