# Eine Architektur für Programmsynthese aus natürlicher Sprache

Mathias Landhäußer[1]

**Abstract:** Diese Dissertation entwirft eine Architektur namens Natural Language Command Interpreter (NLCI), mit der natürlichsprachliche Programme in Quelltext übersetzt werden können. Die Textanalysen sind in einem Fließband angeordnet, wurden jedoch – anders als bei bisherigen Ansätzen – von der gewählten Domäne und der zugehörigen API entkoppelt. Daher sind sie wiederverwendbar. Das nötige Wissen über die API wird in Form eines Modells bereitgestellt. Soll die Domäne (oder die anzusprechende API) gewechselt werden, muss lediglich ein neues Modell bereitgestellt werden.

NLCI wurde in zwei Fallstudien evaluiert: Animationserzeugung und Heimautomation. Die Ergebnisse zeigen, dass NLCI flexibel genug ist, um in beiden Szenarien verwendet zu werden. NLCI erzeugt in vielen Fällen die richtigen Methodenaufrufe ($F_1$=78%) und ermittelt die gewünschte Reihenfolge der Aktionen. Kontrollstrukturen werden mit einer Genauigkeit von 89% synthetisiert.

**Keywords:** Sprachverarbeitung, Programmiersystem, Programmsynthese, Softwaretechnik

„The only way a person can truly concentrate on his problem and solve it without constraints imposed on him by the communication problem are if he is able to communicate directly with the computer without having to learn some specialized intermediate language." Jean E. Sammet, 1966

## 1 Einleitung

Im Jahr 2014 verlautbarte US-Präsident Obama in einem Interview: „Everybody's got to learn how to code early." Ein Jahr zuvor appelierte er an jugendliche US-Amerikaner, sich mit Informatik zu beschäftigen – bereits Grundschulen bieten mittlerweile Programmierkurse an [Pr13]. Obama verschwieg hierbei, dass sich seine eigenen Töchter mit dem Programmieren beschäftigt haben, es aber nicht mögen. Und genau hier liegt der Knackpunkt: Nicht jeder möchte programmieren (lernen). Aktuelle Statistiken besagen, dass lediglich einer von 100 Amerikanern über Programmierkenntnisse verfügt – und das obwohl die USA eines der Länder mit den höchsten Programmiererdichten der Welt sind [MKB06].

Gleichzeitig sind wir von immer mehr programmierbaren Geräten umgeben: Alleine die Zahl der Mobiltelefone wurde bereits 2013 auf 6,8 Mrd. geschätzt – fast eins auf jeden Menschen [In14]. Hinzu kommen Laptops, Fahrzeuge, Haushaltsgeräte usw. Es ist anzunehmen, dass dieser Trend durch das Internet der Dinge noch verstärkt wird. Neue Anwendungen werden die programmierbaren Geräte auf ungeahnte Weise miteinander verbinden – man stelle sich bspw. vor, man befehle einem automatischen Staubsauger „Hör auf zu

[1] Karlsruher Institut für Technologie, Institut für Programmstrukturen und Datenorganisation, Am Fasanengarten 5, 76131 Karlsruhe, mathias.landhaeusser@kit.edu

saugen, wenn ich Musik höre!“ Derartige Befehle sind eine Art Programmierung – selbst wenn es sich in diesem Beispiel nur um ein einzelnes Kommando handelt.

Heutige Programmiertechnologien erlauben es uns nicht, jeden zum Programmierer auszubilden. Die benötigten Kenntnisse und Fähigkeiten sind zu umfangreich und/oder kompliziert für Laien. Die wichtigste Eigenschaft der Computer – ihre Programmierbarkeit – bleibt damit den meisten Menschen verschlossen. Daher schließt sich diese Arbeit der Forderung von Jean E. Sammet an: Um die Programmierbarkeit von Rechnern für Laien zu erschließen, entwirft diese Arbeit ein Programmiersystem, das mit natürlicher englischer Sprache angesteuert wird und das die semantische Lücke zwischen der natürlichen Sprache und Programmiersprachen verringert. Benutzer werden hierdurch in die Lage versetzt, mit geschriebenem Text zu programmieren, ohne die Syntax und Schlüsselwörter von Programmiersprachen vorher erlernen zu müssen.

Die Arbeit zielt darauf ab, Geräte und Anwendungen für Laien programmierbar zu machen [La16]. Sie erschließt so – zu einem gewissen Grad – die Programmierbarkeit von Rechnern für jedermann. Komplexe Systeme, Algorithmik u. Ä. sollen nicht vom Endanwender entwickelt, sondern benutzt werden. Große Systeme werden auch weiterhin von Experten mit den üblichen Programmierkonstrukten entwickelt werden.

Die Arbeit orientiert sich an fünf Thesen, die in ihrem Verlauf beleuchtet werden:

1. Die vorgestellte Architektur kann genutzt werden, um natürlichsprachlich ausgedrückte, imperative Programme in Quelltext zu übersetzen. Endanwender ohne Programmierkenntnisse bilden die Zielgruppe, weswegen der Fokus auf skriptartige Ablaufbeschreibungen und nicht auf komplexe Algorithmen gelegt wird.
2. Es ist dafür nicht notwendig, die zulässige Eingabesprache stark einzuschränken; die Ergebnisse heutiger NLP-Werkzeuge sind als Grundlage gut genug.
3. Das in der Architektur vorgesehene Modell für das Domänenwissen kann ausgetauscht und so die Domäne gewechselt werden. Das Modell kann einfach erzeugt werden (bei geeigneten Quellen automatisch).
4. Unchronologische Beschreibungen können korrekt übersetzt werden: Die Reihenfolge der Beschreibung spielt eine untergeordnete Rolle, solange die zeitlichen Beziehungen zwischen den Aktionen genannt werden. Dann kann die chronologisch korrekte Reihenfolge der Aktionen rekonstruiert werden.
5. Kontrollstrukturen (bspw. einfache Wiederholungen oder Parallelität) können aus natürlichsprachlichen Beschreibungen synthetisiert werden.

## 2 Herangehensweise

Ziel dieser Arbeit ist es, eine domänenunabhängige Architektur zu schaffen, mit der wir der Vision der Programmierung in natürlicher Sprache einen Schritt näher kommen. Kern und Namensgeber der hier vorgestellten und implementierten Architektur ist der Natural Language Command Interpreter, kurz NLCI [LWT16]. Er ist eine sprachliche Schnittstelle für Computersysteme und dafür zuständig, die vom Benutzer ausgedrückten Anweisungen auf Befehle abzubilden, die der darunter liegende Rechner ausführen kann.

Frühere Ansätze des Programmierens in natürlicher Sprache beschränkten sich auf bestimmte Domänen und ermöglichten es so bereits beim Entwurf des Systems die notwendigen Prüfungen zu hinterlegen. Die Erkenntnisse aus den verwandten Arbeiten zeigen, dass eine Programmsynthese aus natürlicher Sprache mit zwei Einschränkungen erreicht werden kann: Man kann 1. den Umfang der erlaubten Eingabesprache einschränken, d.h. die Grammatik und das Vokabular (im Extremfall erhält man so eine „klassische“ Programmiersprache). Man kann 2. das anzusprechende System so stark einschränken, dass damit die sprachliche Komplexität automatisch geringer wird. Frühere Ansätze verweben, unabhängig vom Vorgehen, meist die Analyse der Eingabe mit der Code-Erzeugung bzw. Steuerung derart, dass sie untrennbar oder getrennt voneinander nicht nutzbar sind. Domäneninformationen (d.h. Funktionalität, Daten u. Ä. des anzusprechenden Systems) werden dabei häufig in die Textanalyse mit einbezogen, wodurch die Ergebnisse nicht verallgemeinerbar sind: Ein Domänenwechsel ist gleichbedeutend mit einer Neuentwicklung.

### 2.1 Uneingeschränkte Sprache und Entkopplung von Textanalyse und Ziel-API

Um die Anwendbarkeit von NLCI durch Laien möglichst einfach zu gestalten, schränkt die vorgestellte Arbeit die Eingabesprache nicht ein – weder die Grammatik noch die Wortwahl. Zur Verarbeitung der textuellen Eingabe greift NLCI auf bewährte Werkzeuge der Computerlinguistik zurück, z.B. Stanford CoreNLP zum Zerteilen der Eingabe [Ma14] und WordNet der Universität Princeton für die Erschließung von Synonymen sowie Ober- und Unterbegriffen [Mi95]. Unsere Fragestellung ist keine der Computerlinguistik, z.B. „Wie kann man in einem Satz Subjekt, Prädikat und Objekt bestimmen?“, sondern eine der Softwaretechnik: „Wie können die verfügbaren Ergebnisse der Computerlinguistik so integriert werden, dass Sprachverstehen für das Programmieren ohne Spracheinschränkung möglich ist?“

Neuartig an der vorgestellten Architektur ist die Entkopplung der Textanalyse von den Informationen über das anzusprechende System: Um letztendlich ein Programm erzeugen zu können, benötigt man zwingend Informationen über die zugrundeliegenden Programmierschnittstellen, bspw. die Namen der Methoden. Diese Informationen werden der Textanalyse und der Programmsynthese als Software-Modell (SW-Modell) zugänglich gemacht. So wird die Analyse mit dem SW-Modell konfiguriert und von der anzusteuernden Anwendung unabhängig gemacht. Das Wechseln der anzusprechenden Anwendung erfordert lediglich eine Modellierung der Software (Schritt 1 in Abb. 1). Für die Programmsynthese werden neben der Abbildung von Textelementen auf Programmelemente außerdem Kontrollstrukturen (Wiederholungen u. Ä.) erkannt [LH15] und die Reihenfolge von Aktionen geprüft und korrigiert [LHT14]. Aufgrund der modularen Architektur von NLCI können zudem weitere (ggf. auch domänenabhängige) Analysen eingebunden werden.

Für den Benutzer soll diese Maschinerie möglichst transparent sein – er interagiert textuell in natürlicher Sprache mit der Maschine: Er formuliert sein Programm in natürlicher Sprache (A in Abb. 1), wird in natürlicher Sprache zum Ergänzen und Korrigieren aufgefordert und erhält am Ende das ausführbare Programm bzw. dessen Ergebnis (B in Abb. 1).

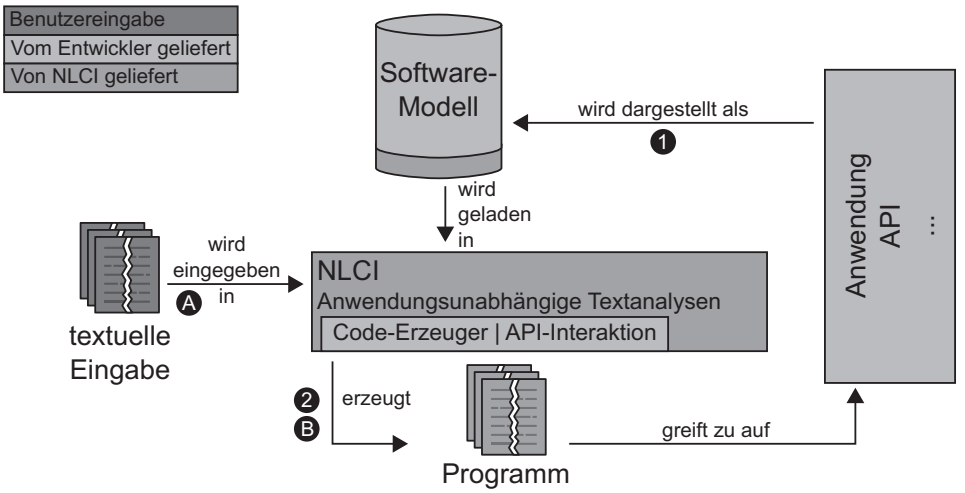


Abb. 1: Überblick über die NLCI-Architektur. Die Verarbeitung soll für den Benutzer transparent geschehen: Er interagiert textuell in natürlicher Sprache mit der Maschine. Für den Entwickler ist Vieles in der Architektur vorgegeben: Er steuert neben der API nur noch eine Beschreibung der API in Form eines SW-Modells und ggf. einen Quelltext-Erzeuger bei; die Struktur des SW-Modells ist dabei von NLCI vorgegeben.

## 2.2 Anbinden von Anwendungen an NLCI

Dem Entwickler wird mit NLCI ein Rahmen gegeben, mit dem sich Anwendungen einfach sprachlich erschließen lassen: NLCI ist eine Architektur, mithilfe derer er Anwendungen um Sprachkompetenz erweitern kann, ohne dass er selbst über Kenntnisse im Bereich der Sprachverarbeitung verfügen muss. Wie Abb. 1 verdeutlicht, muss er lediglich eine Modellierung der Software sowie eine Quelltexterzeugung beisteuern wenn er (s)eine Software an NLCI anbinden und damit sprachlich ansteuerbar machen möchte. An dieser Stelle reicht es aus, sich dieses Modell als Telefonbuch für die Ziel-API vorzustellen. Die dahinter liegende Idee ist der Wunsch, die Ziel-Plattform, also die Domäne, austauschbar zu machen ohne NLCI anpassen zu müssen; das erlaubt letztlich die einfache Anpassung von NLCI auf andere Domänen. Die Erstellung des Modells (Schritt 1 in Abb. 1) kann manuell erfolgen; bei großen APIs ist jedoch eine Automatisierung hilfreich. Eine automatische Modellerzeugung kann einfach implementiert werden, wenn sich die API an gängige Regeln zur Namensvergabe hält (z.B. Substantive für Klassennamen, Verben für Methoden, Verwendung von *CamelCase* usw.). NLCI liefert zudem eine Modell-Schablone, die nur noch mit den Details zur Software gefüllt werden muss: Der Entwickler muss nur beschreiben, welche Elemente seine API bereitstellt.

Die zweite Schnittstelle zwischen NLCI und der Ziel-API (neben dem SW-Modell) ist die Quelltext-Erzeugung: Nach der Verarbeitung des Eingabetextes innerhalb von NLCI werden die Analyseergebnisse ausgewertet und der gewünschte Quelltext ausgegeben (Schritt 2 in Abb. 1). Der Quelltext-Erzeuger muss – ähnlich wie ein Frontend eines Übersetzers – auf die gewünschte Programmiersprache zugeschnitten werden. Er ist un-

abhängig von der anzusteuernden Software und den Textanalysen und folglich ebenfalls wiederverwendbar: Sofern die anzusteuernde API in einer unterstützten Programmiersprache vorliegt, ist nichts weiter zu tun. Existiert keine vorbereitete Anbindung, so muss der Entwickler eine Komponente liefern, die Quelltext in der gewünschten Programmiersprache erzeugen kann. Hierbei kann die Komponente auf die Informationen der Analysestufen zugreifen, ohne dass diese bekannt bzw. verstanden sein müssen. Ein Domänenwechsel und damit das Erschließen neuer Anwendungen und Anwendungsfälle wird somit stark vereinfacht und benötigt keine Anpassungen an NLCI selbst.

### 2.3 Von NLCI bereitgestellte Analysen

Um das von NLCI benötigte SW-Modell auf die Verwendung vorzubereiten, muss die technische API-Beschreibung aufbereitet werden. Überführt man eine objektorientierte API direkt in ein Modell, so fehlen die für die Analysen nötigen sprachlichen Informationen. Daher kann NLCI das Modell automatisch aufbereiten und mit sprachlichen Informationen anreichern. So werden bspw. Bezeichner aufgetrennt, wenn sie aus mehreren Wörtern bestehen und Synonyme aus WordNet hinzugefügt [LWT16]. Diese zusätzlichen Informationen können später in der Textanalyse verwendet werden.

Anschließend wird das Eingabedokument von verschiedenen Analysemodulen verarbeitet und gewonnene Informationen (z.B. welche API-Methoden aufzurufen sind und in welcher Reihenfolge) als Annotation im Dokument hinterlegt. Das letzte Modul wertet diese Annotationen aus und erzeugt den gewünschten Quelltext. In der prototypischen Implementierung von NLCI sind die folgenden Module zur Textanalyse enthalten:

1. Vorverarbeitung des Eingabetextes mit computerlinguistischen Standardwerkzeugen: Eingesetzt werden der Wortartmarkierer und der Zerteiler aus dem Paket CoreNLP der Universität Stanford.
2. Verknüpfung von Text und API: Zu den Textelementen (z.B. Subjekte, Objekte, Prädikate usw.) werden entsprechende API-Elemente ermittelt und mithilfe einer Bewertungsfunktion sortiert [LWT16].
3. Auswahl der Methodenargumente: Basierend auf der zweiten Analyse wählt NLCI Argumente für Methodenaufrufe in den Sätzen aus und bestimmt, welche Methodenaufrufe im Kontext des gesamten Textes generiert werden sollen [LWT16].
4. Korrektur der Reihenfolge: Ist die Beschreibung einer Aktionsfolge nicht chronologisch, so ermittelt NLCI die gewünschte Reihenfolge aus dem Text [LHT14].
5. Erkennung von Kontrollstrukturen: Beschreibungen von Aktionsfolgen enthalten selten explizite Kontrollstrukturen wie Schleifen oder Ähnliches. NLCI leitet daher Kontrollstrukturen aus den sprachlichen Entsprechungen ab [LH15].

Die Analysen sind in einer Fließbandarchitektur zusammengefasst, sodass nachgelagerte Analyseschritte auf die Ergebnisse von vorangegangenen zugreifen können. Zudem ist es möglich, domänenspezifische Analysen hinzuzufügen und so die Qualität des Gesamtergebnisses zu steigern. Für die Evaluation wurden lediglich die oben vorgestellten und keine domänenspezifischen Analysen eingesetzt.

## 3 Evaluation

Die empirische Überprüfbarkeit der entwickelten Werkzeuge ist eine zentrale Anforderung. Daher wurde bereits vor und während der Entwicklung ein Textkorpus aufgebaut. Es sollte die Entwicklung lenken und am Ende zur Evaluation genutzt werden können. Die Texte beschreiben Aktionsfolgen, die von NLCI in ausführbare Programme übersetzt werden sollen. Um möglichst realitätsnahe Eingabetexte zu erhalten, wurden sie von 66 Probanden verfasst, die über unterschiedliche technische Vorkenntnisse verfügen. Die Auswahl der Probanden sollte möglichst breit gestreut sein, ist jedoch pragmatisch gehalten: Doktoranden unseres Lehrstuhls stellen einen Teil der Probanden, Studenten einen weiteren. Sie verfügen über teilweise umfangreiche Programmierkenntnisse und ihre Englischkenntnisse sind recht hoch. Den Rest der Probanden stellen Nicht-Informatiker verschiedener Herkunft. Sie vertreten die Zielgruppe von NLCI. Der überwiegende Teil der Probanden ist Deutsch-Muttersprachler.

Das Vorgehen, NLCI mit realistischen Eingabetexten zu testen, ist methodisch sehr wichtig: Die Problemstellung wurde vor Projektbeginn (möglichst) realitätsnah festgelegt. Während der Forschung und der Implementierungsphase konnte mit echten Problemen gearbeitet werden, anstatt sich künstliche (möglicherweise leichtere oder realitätsferne) Beispiele selbst zu formulieren. Darüber hinaus ermöglicht das Korpus verschiedene Ansätze zur Lösung eines Problems objektiv miteinander zu vergleichen; so können vielversprechende Ansätze schnell identifiziert und schwächere aussortiert werden.

### 3.1 Fallstudien

Der Prototyp von NLCI wurde mit zwei Anwendungen evaluiert. Die erste Anwendung ist openHAB[2], das intelligente Häuser steuert. Es ist ein vergleichsweise einfaches System und der betrachtete Haushalt verfügt über 114 Elemente (Klassen) und lediglich 9 Funktionen (Methoden). Die sprachliche Erschließung einer Haussteuerung ist dennoch ein interessanter Einsatzzweck: NLCI erzeugt aus Befehlen Steuerkommandos, z.B. zum Einschalten eines Lichts. Das benötigte SW-Modell konnte automatisch erzeugt werden und die Erkennung der Befehle funktioniert zufriedenstellend ($F_1$: 71,8%). Diese Fallstudie zeigt, dass NLCI dazu verwendet werden kann, kurze Befehle umzusetzen.

Die zweite Fallstudie betrachtet Alice, eine Animationssoftware mit der man kurze Trickfilme erzeugen kann [Co97]; Alice verfügt über 914 3D-Objekte mit etwa 400 verschiedenen Funktionen und ist damit deutlich umfangreicher als openHAB. NLCI erzeugt hier aus Drehbüchern die Befehle zur Erzeugung von Animationen. Um die Güte der erzeugten Programme zu bestimmen, wurden Drehbücher aus dem NLCI-Korpus verarbeitet und anschließend geprüft, ob korrekte Aufrufe erzeugt wurden. Diese Fallstudie zeigt, dass NLCI auch dazu geeignet ist, skriptartige Programme als Aktionsfolgen zu beschreiben ($F_1$: 72,1%). Die Eingaben der Benutzer können nicht nur auf Methodenaufrufe abgebildet werden, sondern auch auf Kontrollstrukturen [LH15]. Zudem kann die beabsichtigte Reihenfolge der Aktionen in den überwiegenden Fällen rekonstruiert werden [LHT14].

[2] Siehe `http://www.openhab.org/`, zuletzt besucht am 14.02.2017.

### 3.2 Aufbau des Software-Modells

Die in beiden Fallstudien benötigten SW-Modelle sind einfach zu erstellen. Die Modelle wurden in beiden Fällen mit einem Generator aufgebaut, für dessen Implementierung die jeweilige Ziel-API analysiert werden musste. Im Fall von openHAB war dies sehr einfach möglich, da die API eine einfache Struktur aufweist: Der Generator umfasst 370 Codezeilen und konnte in weniger als einem Tag implementiert werden. Die verfügbaren Klassen konnten einfach aus der openHAB-Konfigurationsdatei eingelesen werden. Alice verfügt über eine komplizierte Struktur, weswegen der Generator mit 2700 Codezeilen umfangreicher ausfällt; er konnte in vier Wochen implementiert werden.

### 3.3 Zusammenfassung der Evaluation

Die Ergebnisse der Programmsynthese sind auch ohne Einschränkung der Eingabesprache zufriedenstellend. NLCI kann Imperative und Aussagesätze verarbeiten, solange diese richtig von den verwendeten NLP-Werkzeugen verarbeitet werden. Dabei kommt es NLCI nicht darauf an, ob die Sätze im Aktiv oder Passiv geschrieben sind und ob sie Nebensätze enthalten. Da viele Analysen auf typisierten Abhängigkeitsgraphen basieren, ist die Reihenfolge der Wörter im Satz ebenfalls unerheblich (bspw. kann vom Standardmuster „Subjekt, Prädikat, Objekt“ abgewichen und Nebensätze eingeflochten werden).

Da die Ergebnisse der Analysen von den Ergebnissen der computerlinguistischen Werkzeuge abhängen, sind sie empfindlich gegenüber falschen Zwischenergebnissen. Die meisten Fehler in den erzeugten Programmen sind jedoch keine falsch übersetzte Aktionen, sondern welche, die nicht übersetzt werden konnten. Bei der Interpretation der Ergebnisse muss berücksichtigt werden, dass die computerlinguistischen Werkzeuge nicht speziell auf diesen Einsatzzweck vorbereitet wurden. Insofern sind die Ergebnisse als untere Schranke zu betrachten. Insbesondere das Fehlen von Aktionen aufgrund einer mangelhaften Verb-Erkennung scheint einfach durch ein Training der verwendeten Werkzeuge lösbar zu sein.

## 4 Zusammenfassung

Die vorgestellte Architektur entkoppelt das Wissen über die anzusteuernde API, d.h. das Domänenwissen, von der Textanalyse mithilfe eines Modells. So kann erreicht werden, dass die Textanalysen wiederverwendbar sind, auch wenn sich die Ziel-API oder die Ziel-Programmiersprache ändert. Die prototypische Implementierung von NLCI gibt eine Struktur für das benötigte SW-Modell vor, die auf objektorientierte Systeme zugeschnitten ist und einfach gefüllt und erweitert werden kann. (Ein SW-Modell kann für kleine Systeme sogar manuell aufgebaut werden.)

Die Textanalysen sind durch eine Fließbandarchitektur ebenfalls voneinander entkoppelt. Die Ergebnisse der (Vor-)Verarbeitung stehen allen folgenden Analysestufen zur Verfügung. Eine Stufe kann die Ergebnisse einer vorangehenden Stufe verwenden, interpretieren und sogar korrigieren oder erweitern. Die vorgestellten Analysen wurden

als domänenunabhängige Module entworfen. Es ist jedoch möglich, domänenspezifische Analysen in das Fließband einzuhängen, um so die Leistungsfähigkeit des Gesamtsystems zu verbessern.

Die Fallstudien mit NLCI haben gezeigt, dass die vorgestellte Architektur und die entworfenen Analysen dazu geeignet sind, aus natürlichsprachlichen Beschreibungen oder Befehlen Skripte bzw. API-Aufrufe zu erzeugen. Die Auswertung der Ergebnisse hat aber auch gezeigt, wo die größte Schwäche von NLCI liegt: Fehler, die von den computerlinguistischen Analysen in der Vorverarbeitung gemacht werden, können nur bedingt entdeckt und nicht automatisch ausgeglichen werden.

Die vorliegende Arbeit suchte anhand der Fallstudien mit NLCI nach Belegen für die eingangs formulierten Thesen: Die beiden Fallstudien belegen, dass NLCI sowohl für openHAB als auch für Alice in der Lage ist, die natürlichsprachlichen Beschreibungen in Quelltext zu übersetzen (These 1). NLCI ermittelt in vielen Fällen die gewünschten Methodenaufrufe ($F_1 = 78,3\%$).

NLCI kann Imperative und Aussagesätze verarbeiten, solange diese richtig von den verwendeten NLP-Werkzeugen verarbeitet werden (These 2). Zur Evaluation von NLCI wurden Eingabetexte verwendet, die von Probanden geschrieben wurden. Der überwiegende Teil der Eingabesätze kann mit den verfügbaren NLP-Werkzeugen korrekt verarbeitet werden; treten Fehler auf, ließen sie sich häufig darauf zurückführen, dass die verwendeten NLP-Werkzeuge nicht speziell für den Anwendungsfall trainiert wurden und nicht auf systematische oder konzeptionelle Unmöglichkeit.

Das in der Architektur vorgesehene SW-Modell für das Domänenwissen kann ausgetauscht und so die Domäne gewechselt werden (These 3). Die Fallstudien zeigen, dass die entworfenen Analysen mit Beschreibungen für beide Szenarien umgehen können. Die nötigen Modelle konnten ohne Anpassungen an der Vorlage von NLCI erstellt und automatisch aufgebaut werden. Das Modell kann einfach erzeugt werden (bei geeigneten Quellen, wie denen in den Fallstudien, sogar automatisch).

Die Evaluation der Reihenfolgenkorrektur hat gezeigt, dass chronologisch nicht in der richtigen Reihenfolge genannte Aktionen erkannt werden können, wenn die Beschreibung Temporalausdrücke enthält. NLCI ist häufig in der Lage, die gewünschte Reihenfolge zu rekonstruieren und so eine korrekte Aktionsfolge zu erstellen (These 4): 86% der Temporalausdrücke können korrekt ausgewertet werden und ein großer Teil der übrigen führt nicht zu Fehlern bei der Feststellung der gewünschten Reihenfolge.

Die Evaluation der Kontrollstrukturanalyse hat gezeigt, dass man den Benutzern eines natürlichsprachlichen Programmiersystems nicht vorschreiben muss, wie Kontrollstrukturen diktiert werden müssen (These 5). Wie einige verwandte Arbeiten nahelegten, lassen sich viele Kontrollstrukturen aus natürlichsprachlichen Formulierungen ableiten und entsprechend umsetzen. Die Synthese der betrachteten Kontrollstrukturen rundet die Fähigkeiten von NLCI ab, die in 89% der Fälle funktioniert; geht man von einer korrekten computerlinguistischen Vorverarbeitung aus, kann die Erkennungsrate sogar auf 97% gesteigert werden.

Die Erkenntnisse aus der Forschungsarbeit und die Ergebnisse der Evaluation zeigen, dass heute Programmierung in natürlicher Sprache möglich ist und die Eingabesprache nicht künstlich eingeschränkt werden muss. Dem übergeordneten Ziel, Benutzern zu ermöglichen in natürlicher Sprache zu programmieren, ist NLCI ein großes Stück näher gekommen. Die obigen Belege für die Thesen zeigen, dass das angestrebte Ziel in den betrachteten Einsatz-Szenarien erreicht werden konnte. Die Evaluation des Prototyps hat gezeigt, dass NLCI flexibel genug ist, um in diesen stark unterschiedlichen Einsatz-Szenarien verwendet zu werden. Die nötige Einschränkung der Domäne ist kein Nachteil, da die Akquise neuer Domänen einfach vonstatten geht. Auch wenn es auf dem Gebiet der natürlichsprachlichen Programmierung noch viele offene Fragen gibt, eröffnet diese Arbeit viele neue Perspektiven.

## 5 Ausblick

Trotz – oder gerade: aufgrund – der erfreulichen Ergebnisse von NLCI in dieser Arbeit ergeben sich weitere Forschungsfragen, die untersucht werden müssen. Die folgenden Abschnitte skizzieren mögliche nächste Schritte auf dem Weg zu einem noch besseren, und vor allem interaktiven natürlichsprachlichen Programmiersystem:

Die Arbeiten zur Korrektur der Reihenfolge und der Erkennung der Kontrollstrukturen haben gezeigt, dass es kein (gut funktionierendes) Verfahren gibt, das Korreferenzketten für Aktionen aufbauen kann. Ansätze aus dem Bereich der Auflösung von Personalpronomina könnten weiterentwickelt werden, um tatsächlich identische (nicht gleiche) Aktionen in einer Aktionsfolge zu ermitteln. Da das SW-Modell hilft, verschiedene Paraphrasierungen derselben Aktion auf einen Methodenaufruf abzubilden, solle es auch helfen, Aktionsreferenzen auszuwerten.

Das SW-Modell enthält derzeit weder Vor- oder Nachbedingungen noch Invarianten. Wären diese Informationen verfügbar, könnten sie für die Sicherstellung der korrekten Reihenfolge genutzt werden. Zudem könnte aus einer unterbrochenen Kette von Vor- und Nachbedingungen zu einer Aktionsfolge geschlossen werden, dass eine Aktion fehlt und an welcher Stelle. Ob dieses zusätzliche Wissen in der Praxis verfügbar und nutzbar ist, muss untersucht werden.

NLCI beschränkt sich auf die Analyse geschriebener Sprache. Tichy und Weigelt entwerfen ein vielversprechendes System, das gesprochene Sprache verarbeiten können soll [WT15]. Ihr System ist nicht als Fließband aufgebaut, sondern als Agentensystem, wobei die Agenten den Verarbeitungsstufen von NLCI entsprechen.

Anwendungsgebiete für NLCI ergeben sich in vielen Bereichen. Mit den Analysen könnten Arbeitsanweisungen für (Haushalts-)Roboter formuliert werden und die Analyse paralleler Aktionsfolgen könnte für die Modellierung von Arbeitsabläufen genutzt werden. Zudem ist es denkbar, dass alle Programme, die derzeit über eine grafische Oberfläche gesteuert werden, sprachliche Befehle empfangen. Eine weitere aussichtsreiche Anwendung ist das Erzeugen von Testfällen durch Mitarbeiter von Fachabteilungen.

## Literaturverzeichnis

[Co97] Conway, Matthew J.: Alice: Easy-to-Learn 3D Scripting for Novices. Phd thesis, Faculty of the School of Engineering and Applied Science, University of Virginia, 1997.

[In14] International Telecommunication Union (ITU): The World in 2014: ICT Facts and Figures. Mai 2014.

[La16] Landhäußer, Mathias: Eine Architektur Für Programmsynthese Aus Natürlicher Sprache. KIT Scientific Publishing, Karlsruhe, 2016.

[LH15] Landhäußer, Mathias; Hug, Ronny: Text Understanding for Programming in Natural Language: Control Structures. In: Proceedings of the 4th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering. Mai 2015.

[LHT14] Landhäußer, Mathias; Hey, Tobias; Tichy, Walter F.: Deriving Timelines from Texts. In: Proceedings of the 3rd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering. S. 45–51, Juni 2014.

[LWT16] Landhäußer, Mathias; Weigelt, Sebastian; Tichy, Walter F.: NLCI: A Natural Language Command Interpreter. Automated Software Engineering, August 2016.

[Ma14] Manning, Christopher D.; Surdeanu, Mihai; Bauer, John; Finkel, Jenny; Bethard, Steven J.; McClosky, David: The Stanford CoreNLP Natural Language Processing Toolkit. In: Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations. Association for Computational Linguistics, Baltimore, Maryland, S. 55–60, Juni 2014.

[Mi95] Miller, George A.: WordNet: A Lexical Database for English. Commun. ACM, 38(11):39–41, November 1995.

[MKB06] Myers, Brad A.; Ko, Andrew J.; Burnett, Margaret M.: Invited Research Overview: End-User Programming. In: CHI '06 Extended Abstracts on Human Factors in Computing Systems. CHI EA '06, ACM, New York, NY, USA, S. 75–80, 2006.

[Pr13] President Obama Asks America to Learn Computer Science, Dezember 2013. `https://www.youtube.com/watch?v=6XvmhE1J9PY`.

[WT15] Weigelt, Sebastian; Tichy, Walter F.: Poster: ProNat: An Agent-Based System Design for Programming in Spoken Natural Language. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE). Jgg. 2, S. 819–820, Mai 2015.

**Mathias Landhäußer** wurde am 02. Juni 1983 geboren. In seiner Forschung beschäftigt er sich damit, wie computerlinguistische Werkzeuge im Softwareentwicklungsprozess eingesetzt werden können, insbesondere wenn es um kreative Aufgaben wie Programmierung geht. Zudem interessiert er sich für empirische Forschungsmethoden in der Softwaretechnik. Er promovierte im Jahr 2016 am Karlsruher Institut für Technologie (ehemals Universität Karlsruhe (TH)). Zuvor studierte er – ebenfalls am KIT – Informationswirtschaft. Er ist Mitglied der ACM und der GI. Mehr über Mathias erfahren Sie auf seinem LinkedIn-Profil unter `https://www.linkedin.com/in/mathiaslandhaeusser`.