Eine Programmiersprache zur souveränen Datenverarbeitung

Fabian Bruckner[1] · Ralf Nagel[1] · Dominik Krüger[1]
Sven Wenzel[1] · Boris Otto[1,2]

[1]Fraunhofer ISST – Digitization in Service Industries
{fabian.bruckner | ralf.nagel | dominik.krueger | sven.wenzel
boris.otto}@isst.fraunhofer.de

[2]TU Dortmund, Lehrstuhl für Supply Net Order Management
boris.otto@tu-dortmund.de

Zusammenfassung

Zur Addressierung der systemkritischen Eigenschaft der digitalen Souveränität im Kontext der Industrie 4.0 wird aktuell die domänenspezifische Programmiersprache D° (gesprochen di'grē) entwickelt, welche für die Verarbeitung von Daten konzipiert ist. Besonderheiten der Sprache sind unter anderem erweiterbare Typen- und Aktivitätssysteme, welche wesentlich stärker abstrahieren als es gängige Programmiersprachen tun. Darüber hinaus verfügt D° über integrierte Mechanismen für Usage Control und überprüft Bedingungen ohne dynamischen Anteil bereits zur Übersetzungszeit. Die verbleibenden Nutzungsbedingungen werden zur Laufzeit von der Ausführungsumgebung durchgesetzt. Hierdurch ist es nicht mehr notwendig, zur Durchsetzung von Nutzungsbedingungen eine zusätzliche Software zu verwenden und zu betreiben. Darüber hinaus sind keine Modifikationen der Applikationen notwendig, um die Lösungen verwenden zu können. Stattdessen bietet D° eine ganzheitliche und somit einfacher zu nutzende Lösung. Diese Arbeit skizziert den Aufbau von D° und stellt einige besondere Aspekte und entwickelte Konzepte detailliert vor.

1 Einleitung

Die Bedeutung von Daten wird im Kontext der Industrie 4.0 immer wichtiger, da ein zentraler Aspekt die Verarbeitung von großen Datenmengen zur Erlangung nützlicher Informationen ist [LKY14]. Hierdurch entwickeln sich Daten immer mehr zu einem eigenen strategischem Asset [Ott15] und es ergibt sich die Notwendigkeit, die digitale Souveränität an diesen Assets zu wahren [PGB11, OCM+16]. Der Begriff der digitalen Souveränität bzw. Datensouveränität beschreibt die Fähigkeit eines Unternehmens oder einer Person, selber über die Nutzung von eigenen Daten entscheiden zu können und Regelungen zur technischen Durchsetzung festzulegen. Zu diesem Zweck definieren Datenbesitzer Nutzungsbedingungen (Policies) für die Daten, welche festlegen, in welchem Umfang die Daten durch bestimmte Personen bzw. Unternehmen genutzt werden dürfen. Da nach juristischem Verständnis kein Eigentum an Daten möglich ist [Jlu16, Dor14], wird der Begriff (Daten-) Besitzer im Folgenden synonym mit Rechteinhaber verwendet.

Ein weiterer wichtiger Punkt ist, dass viele kleine und mittelständische Unternehmen keine

P. Schartner · N. Pohlmann (Hrsg.) · D•A•CH Security 2018 · syssec (2018) 35-46.

eigene IT-Infrastruktur betreiben, sondern vollständig auf Cloud-Dienste setzten [CZ12], und somit auf einen sachgemäßen Umgang mit den eigenen Daten durch Dritte angewiesen sind. Ebenso werden Daten nicht nur vom Besitzer für eigene Zwecke verarbeitet. Stattdessen kann es gewünscht sein, dass es Partnern ermöglicht wird, bestimmte Berechnungen auf den eigenen Daten durchzuführen, ohne dabei den kompletten Datensatz offenzulegen. Es besteht somit die Anforderung, dass die Daten im eigenen Unternehmen verbleiben. In jedem Fall ist es notwendig, dass die Nutzungsbedingungen, welche der Datenbesitzer definiert, eingehalten werden.

Um diese Nutzungsbedingungen für die Daten durchzusetzten werden Usage Control Mechanismen benötigt. Da Usage Control kein integrierter Bestandteil von gängigen Programmiersprachen ist, müssen existierende Applikationen modifiziert werden um die Lösungen integrieren zu können und bei neuen Applikation muss die Integration direkt von Anfang an geplant werden. Dabei wird Expertenwissen und Vertrauen in die fehlerfreie und korrekte Implementierung der Lösung benötigt.

Diese Lücke soll durch eine domänenspezifische Programmiersprache (DSL) geschlossen werden, deren Zieldomäne die Datenverarbeitung ist und die über integrierte Usage Control Mechanismen verfügt. Der Arbeitstitel für die Sprache lautet D°. Jede Applikation, die mit der Sprache entwickelt wird, verfügt automatisch über integrierte Usage Control Mechanismen, welche die Definition und Durchsetzung von Nutzungsbedingungen erlauben. Anwender der Sprache müssen lediglich ihre Nutzungsbedingungen für die zu verarbeitenden Daten definieren (Datenbesitzer) und die benötigten Berechtigungen einer Applikation festlegen (Entwickler). Hierdurch ist es möglich, bereits zur Übersetzungszeit statische Nutzungsbedingungen, welche keine Abhängigkeit von der Laufzeit haben, bezüglich ihrer Einhaltung zu überprüfen und im Konfliktfall eine Ausführung zu verhindern. Beispiele für solche Nutzungsbedingungen werden in Abschnitt 3.1.2 gegeben. Für den verbleibenden Teil der Nutzungsbedingungen, welcher nicht während des Übersetzungsprozesses geprüft werden kann, wird die Einhaltung zur Laufzeit sichergestellt. Dabei werden die Mechanismen zur Prüfung und Umsetzung nicht parallel und losgelöst von der Applikation betrieben, sondern sind untrennbar in die Ausführungsumgebung verwoben.

D° kann auf unterschiedliche Anwendungsdomänen zugeschnitten werden. Daher sind die Schlüsselworte (bis auf einen kleinen Teil) nicht festgelegt sondern werden aus zentralen Repositories nachgeladen. Diese austauschbaren Schlüsselworte referenzieren zum einen die in D° verwendeten Datentypen und zum anderen die dazu passenden Aktivitäten.

Um Compiler und Parser nicht nach jeder Änderung neu generieren zu müssen, werden diese Schlüsselworte in der D°-Grammatik als Identifier behandelt, auch wenn diese Sprachelemente innerhalb von D° atomar und nicht weiter zerlegbar sind. Ergänzend ist die Komposition neuer Typen oder Aktivitäten innerhalb von D° vorgesehen. Neben den zentralen Repositories für Datentypen und Aktivitäten sind lokale Erweiterung durch die Anwender geplant. Eine Aktivität in D° entspricht einer in sich vollständigen funktionalen Einheit, welche über mehrere Ein- und Ausgaben verfügen kann. Der Begriff ist am ehesten vergleichbar mit dem der Methode, welcher aus der objektorientierten Programmierung bekannt ist.

Durch die Verwendung einer speziellen IDE für D° ist eine Unterstützung der Entwickler wie bei gängigen Programmiersprachen möglich. Für den einzelnen Entwickler verhält sich D° wie eine Programmiersprache mit festen Schlüsselworten.

D° enthält Funktionen, welche die Übermittlung, Übersetzung und Ausführung von Applikationen, sowie die anschließende Rücksendung der Ergebnisse erlauben. Diese Funktionen werden

unter dem Begriff *Remote Processing* zusammengefasst. Hierdurch wird es möglich, dass die Daten im Unternehmen verbleiben und Partner Berechnungen, welche den Nutzungsbedingungen nicht widersprechen, auf den Daten durchführen.

Ein weiteres Merkmal der Sprache ist der im Vergleich zu gängigen Programmiersprachen wesentlich höhere Abstraktionsgrad. Statt den gebräuchlichen primitiven Datentypen (`int`, `long`, `boolean`, ...) gibt es wesentlich fachlichere Datentypen (beispielsweise `Postadresse`), welche über das erweiterbare Typsystem fester Bestandteil der Sprache sind. Dies trifft ebenfalls auf die Aktivitäten der Sprache zu. Statt einer Aktivität zur Konkatenation von Strings, existiert eine atomare Aktivität zur Gültigkeits- & Konsistenzprüfung von Postadressen.

Nachfolgend stellt Kapitel 2 relevante verwandte Arbeiten vor. In Kapitel 3 werden die einzelnen Komponenten strukturiert vorgestellt. Kapitel 4 zeigt anhand eines Beispiels die bisher umgesetzten Konzepten in ihrem aktuellen Stand, welche abschließend in Kapitel 5 zusammengefasst werden.

2 Verwandte Arbeiten

Beim Industrial Data Space handelt es sich um ein Forschungsprojekt, welches vom Bundesministerium für Bildung und Forschung finanziert wird und von 12 kooperierenden Fraunhofer Instituten, bei starker Einbeziehung der Industrie, durchgeführt wird. Das Ziel des Projektes ist die Schaffung eines virtuellen Datenraumes für den souveränen Datenaustausch [OJS+16]. Alle hierzu notwendigen Komponenten und Rollen sind in einer Referenzarchitektur definiert [OLA+17] und in einer zweiten Version verfeinert [Oea18] worden. Da sich das Projekt auf den souveränen Datenaustausch fokussiert, ist keine Ausführungsumgebung für das Projekt definiert worden. D° wäre ein geeigneter Kandidat, um als Programmiersprache im Industrial Data Space verwendet zu werden und dabei den Ansprüchen an die digitale Souveränität zu genügen.

Es existieren diverse Lösungen, welche die Integration von Usage Control Mechanismen in Applikationen erlauben. Im Rahmen der Entwicklung von D° werden existierende Lösungen für Usage Control evaluiert und sofern möglich in die Sprache integriert. In diesem Zusammenhang wird beispielsweise der Ansatz der *sticky policies* evaluiert, welcher die Nutzungsbedingungen direkt an die Daten heftet und anschließend Daten und Policy gemeinsam verarbeitet [Tan08, PCM11, MPB03, TS12]. Weitere Lösungen für Usage Control, welche aus dem Industrial Data Space bekannt sind und für die Verwendung in D° evaluiert werden müssen, sind LUCON [SBW18] und IND^{2}DUCE [SKJH16].

Es existiert eine große Anzahl von domänenspezifischen Programmiersprachen, welche viele verschiedene Domänen adressieren [VDKV+00]. Dabei existieren auch Sprachen, welche auf die Datenverarbeitung abzielen, beispielsweise *PigLatin* [ORS+08], *Jet* [AJRO12], *BDL* [HHKW77], *PADS* [FG05] und *LUSTRE* [HCRP91]. Alle diese Sprachen setzten ihren Fokus auf die Verarbeitung von Daten, sind aber aus verschiedenen Gründen ungeeignet für die abstrakte Abbildung von Datenverarbeitungsprozessen. Beispielsweise setzt BDL starr auf die Definition von Formularen und die automatische Erzeugung von Oberflächen für selbige und sowohl LUSTRE als auch PigLatin weisen nicht den gewünschten Abstraktionsgrad auf. Andere der verfügbaren Sprachen sind für die allgemeine Datenverarbeitung wiederum zu spezifisch, da die verarbeitbaren Daten zu sehr eingeschränkt werden, bspw. ist PADS nur für Ad-hoc-Daten (Logdaten) geeignet. Darüber hinaus setzten die Sprachen, wie beispielsweise

Jet, häufig auf eine technologische Grundlage auf und sind somit nicht für den Einsatz in allen Umgebungen mit arbiträren Persistenztechnologien geeignet.

3 Aufbau der Sprache

Neben den bereits in der Einleitung aufgezeigten Unterschieden zu den gängigen Programmiersprachen existieren weitere Aspekte und Unterschiede, welche D° maßgeblich definieren.

Anders als gängige Programmiersprachen oder beispielsweise BSD verfügt D° nicht über Möglichkeiten zur Erstellung von (graphischen) Oberflächen oder Nutzerinteraktionen. Der Fokus der Sprache liegt auf der automatisierten Verarbeitung von Daten, weswegen solche Funktionen nicht als Kernfunktionalität benötigt werden. Sollte sich die Notwendigkeit für solche Funktionen ergeben, können Schnittstellen definiert werden, die zur Nachrüstung der benötigten Aspekte als externe Erweiterung verwendet werden können.

Ebenso verfügt die Sprache über keinerlei Funktionen, um Programmcode zur Laufzeit dynamisch nachzuladen oder den aus Java oder C# bekannten Reflection Mechanismus zu verwenden. Denn sobald solche Funktionen zur Verfügung stehen, sind die Überprüfungen der Policies, welche zur Übersetzungszeit stattgefunden haben, hinfällig und auch die Durchsetzung von Policies zur Laufzeit wird ungleich schwerer bis unmöglich, da beliebiger Code ausgeführt werden kann.

D° ist eine statisch getypte Programmiersprache, wodurch bereits zur Übersetzungszeit fehlerhafte Typzuweisungen entdeckt werden können. Da Typinformationen mit einbezogen werden können, wird die Menge der Policyanweisungen, welche zur Übersetzungszeit überprüft werden können, erhöht.

Nachfolgend werden die einzelnen Komponenten von D° genauer betrachtet und wichtige Aspekte herausgestellt. Eine Übersicht über alle Komponenten kann Abbildung 1 entnommen werden. Die Abbildung ist unterteilt in die drei Ebenen Spezifikation, Entwicklung und Ausführung. Hierdurch entsteht eine Gruppierung von zusammenhängenden Komponenten.

3.1 Spezifikation

Die Spezifikation der Sprache setzt sich aus mehreren, teils erweiterbaren Modulen zusammen, welche in ihrer Gesamtheit die Programmiersprache D° ausmachen. Das einzige unveränderliche Modul ist das *Grammar*-Modul, welches die Form und Struktur von Applikationen festlegt und definiert wie die Beschreibung neuer Datentypen, Aktivitäten & Policies gestaltet sein müssen.

Die verbleibenden Module (*Typesystem*, Definitionen von *Activities* & *Policies*) sind eine Menge von Ausprägungen der entsprechenden Teile der Grammatik. Zentrale und/oder lokale *Repositories* werden verwendet, um die Basismodule mit Erweiterungen zu versehen.

Bei den Repositories handelt es sich um sensible Infrastruktur. Gelingt es einem Angreifer manipulierte Aktivitäten an Nutzer zu verteilen, ist es möglich die Usage Control Mechanismen auszuhebeln. Aus diesem Grund ist der schreibende Zugriff auf Repositories auf den Betreiber und von ihm ausgewählte Personen bzw. Unternehmen beschränkt. Eine Einschränkung des lesenden Zugriffs durch verschiedene Maßnahmen (beispieslweise Autorisierung & rollenbasierter Zugriff) ist möglich und kann den jeweiligen Anforderungen angepasst werden.

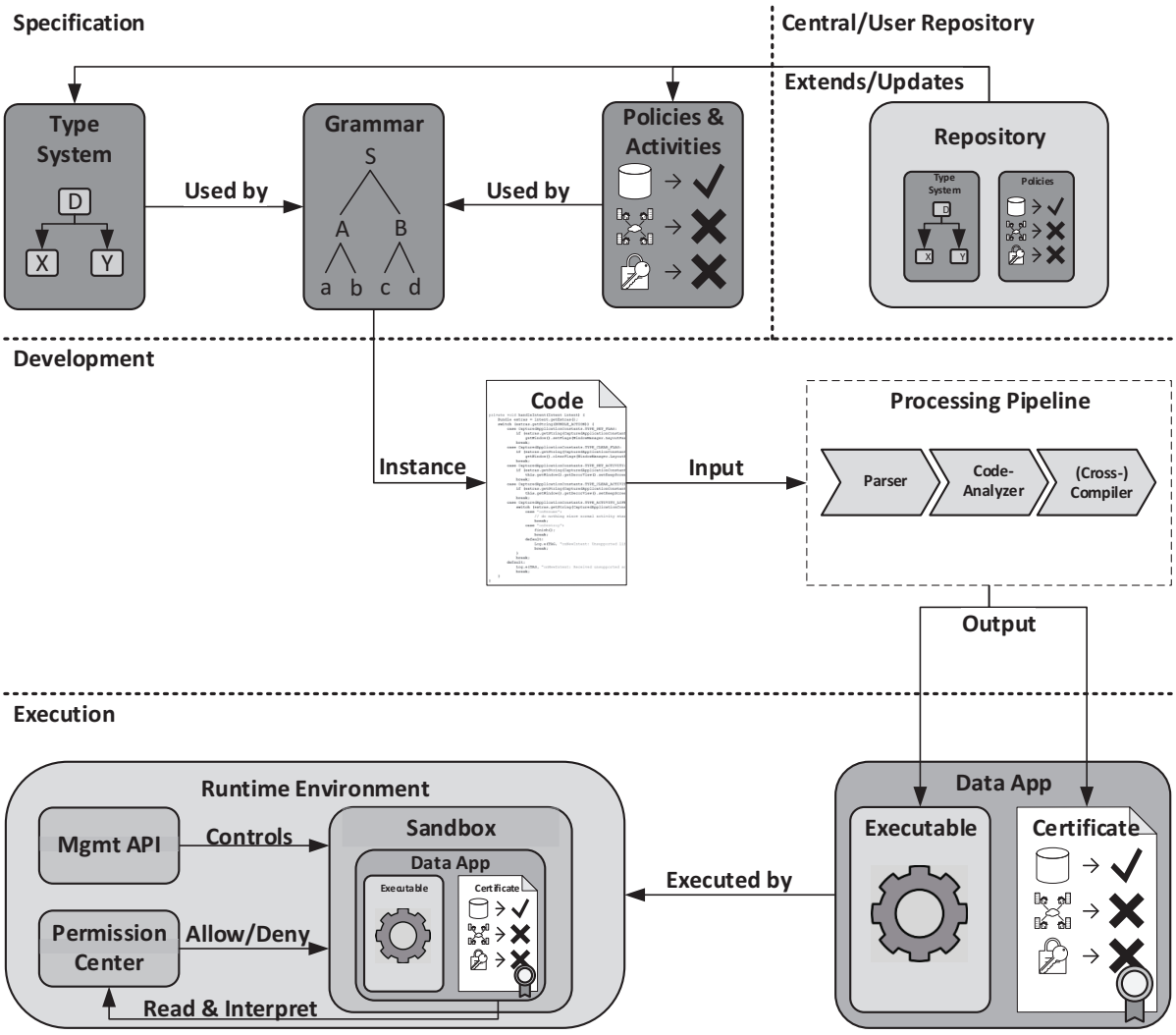

Abb. 1: Schematische Darstellung der verschiedenen Komponenten von D°

Um Namenskonflikte zwischen verschiedenen Erweiterungen und dem Basismodul zu verhindern werden Namensräume zur eindeutigen Identifikation verwendet.

Das Typsystem erlaubt es, primitive und komplexe Datentypen und deren Beziehungen untereinander zu definieren. Ein Schwerpunkt der Typen bildet die Validierung von Instanz-Werten zur Laufzeit.

Aktivitäten bestehen aus mehreren Bestandteilen. Neben der eigentlichen Funktinalität sind benötigte Berechtigungen und entstehende Auswirkungen auf die bearbeiteten Daten zu definieren. Beispielsweise verändert die Funktion “anonymisiere” eingehende Daten (Berechtigung) und macht Daten unkenntlich (Auswirkung). Die Definition von Berechtigungen und Auswirkungen ist ein Teil der Nutzungsbedingungen, welche in Abschnitt 3.1.2 genauer beschrieben werden.

3.1.1 Erweiterbares Typsystem

Ein wichtiger Aspekt von D° ist die Nutzung unterschiedlicher Datentypen bzw. Domänenmodelle. Dabei liegt der Fokus nicht auf maschinen-orientierten Datentypen wie `Character`, `Integer`, `Boolean` oder `String`, sondern in fachlichen Domänentypen wie z.B. `Avise`, `Rechnung` oder `Artikel` aus der Logistik-Domäne. Da D° unabhängig von der fachlichen Domäne konzipiert wurde, müssen die verwendeten Domänenmodelle austauschbar, erweiterbar und anpassbar sein.

Grundlegend wird ein Typ in D° anhand mehrerer Grundeigenschaften definiert. Zum einen muss eine Speicher-Repräsentation festgelegt werden. Diese legt fest, wie der aktuelle Wert einer Instanz des jeweiligen Typs im Hauptspeicher repräsentiert wird. Weiterhin besitzt jeder Typ einen Validator, der bei allen Schreibzugriffen die Gültigkeit des neue Werts vorab prüft und gegebenenfalls die Änderung blockiert. Validatoren sind einfache Java-Klassen, die über eine `accept()`-Methode die Prüfung vornehmen und Fehler als `RuntimeException` melden. Ein simpler Validator ist der `RegExValidator`, der anhand eines regulären Ausdrucks die Gültigkeit prüft. Die Komposition neuer Validatoren über boolesche Operatoren (z.B. `UnionValidator`) ist sehr einfach möglich. Der `ScriptValidator` bietet die Möglichkeit, die Evaluierung in einer Scriptsprache wie JavaScript vorzunehmen. Validatoren werden automatisch über den Klassenpfad von Java erkannt und berücksichtigt.

Für die Instanziierung von Variablen eines Typs in D° wird der initiale Wert festgelegt. Dieser Wert muss zwingend vom festgelegten Validator akzeptiert werden. Der initiale Wert ist im aktuellen Stand noch ein statischer Wert, in einer geplanten Erweiterung wird es auch möglich sein hier analog zum Validator einen Initiator zu definieren, der den initialen Wert berechnet. Damit wird es möglich z.B. ein Bestelldatum mit dem aktuellen Datum vorzubelegen.

Zusätzlich unterstützt das Typsystem von D° Typumwandlungen (Typecasting) über Spezialisierung und Generalisierung. Pro Typ sind mehrere Generalisierungen zulässig. Kreisstrukturen werden allerdings nicht unterstützt. Mit Hilfe dieser Beziehungen wird festgelegt, ob ein Typ in einen Anderen überführt werden kann. Technisch wird beim Schreibzugriff nicht nur der Validator des jeweiligen Typs sondern auch jeder Validator der angegebenen Generalisierungen ausgeführt um dies jederzeit sicherzustellen.

Aus diesen ein-elementigen Typen, die Primitive genannt werden, können über Komposition komplexe Typen abgeleitet werden. Solche Typen werden Composite genannt. Ein Composite-Typ ist eine Key-Value Struktur, die eindeutigen Keys oder Attributen einen spezifischen Type als Value zuweist. Es entstehen Strukturen wie in Pascal (Record), C (Struct) oder Java (Bean). Einem Composite-Typ ist ein zusätzlicher Validator zugeordnet, der für die Konsistenz zwischen mehreren Attributen oder status-abhängigen Prüfungen genutzt werden kann. Die Erweiterung des Typsystems um Arrays, Listen und Maps ist ebenso vorgesehen.

Das Typsystem wird durch eine Referenz-Implementierung in Java realisiert. Variablen oder Instanzen können so in D° repräsentiert werden. Diese Instanzen werden zur Laufzeit an die jeweiligen Aktivitäten übergeben. Handelt es sich um Java-Aktivitäten ist eine direkte Übergabe als Java-Instanz möglich. Werden Aktivitäten aus anderen Quellen oder Programmiersprachen verwendet, werden die Instanzen über ihre Serialisierungen übergeben. Standardmäßig unterstützt das Typsystem hierfür JSON, XML und YML. Durch entsprechende Schnittstellen können neue Textformate leicht ergänzt werden. Weiterhin ist geplant über diesen Weg eine Integration von SQL für relationale Datenbanken zu realisieren.

3.1.2 Nutzungsbedingungen

Bei den Nutzungsbedingungen müssen drei verschiedene Arten bzw. Quellen berücksichtigt werden. Zum einen sind die zu verarbeitenden Daten mit Bedingungen, zum Beispiel in Form von sogenannten Sticky Policies (siehe Abschnitt 2), verknüpft. Diese sagen beispielsweise aus, wie oft auf die Daten zugegriffen werden darf oder dass die Daten nur anonymisiert weitergegeben werden dürfen. Zum anderen spezifiziert der Ausführende, welche Berechtigungen der Anwendung zur Laufzeit gewährt werden. Hier seien beispielhaft die Einschränkungen von Dateizugriffen auf bestimmte Verzeichnisse und Dateien mit bestimmten Größen, Beschränkungen des Netzwerkzugriffs sowie die Erlaubnis, bestimmte Aktivitäten ausführen zu dürfen, genannt. Des Weiteren wird bei der Definition von Aktivitäten festgelegt, welche Berechtigungen diese zur Ausführung benötigt (zum Beispiel „Dateien aus einem festen Pfad lesen“) und wie sie sich auf die verarbeiteten Daten auswirkt (zum Beispiel „Daten wurden anonymisiert“). Die Berechtigungen sollen dabei möglichst feingranular, modular und konfigurierbar sein. Während Betriebssysteme beispielsweise bei Schreibzugriffen nur den Dateipfad prüfen, sollen in D° auch Beschränkungen des Inhalts der Datei möglich sein.

Da alle diese Informationen über die Berechtigungen bereits zur Kompilierzeit der D°-Anwendung vorliegen, kann bereits hier eine erste Prüfung erfolgen. Erfordert eine Applikation beispielsweise, dass Daten nur nach einer vorhergehenden Anonymisierung persistiert werden dürfen, kann dies unter Umständen bereits überprüft werden. Sofern alle Ausführungspfade die eine Aktivität zur Persistierung enthalten zuvor eine „anonymisieren“-Aktivität enthahlten ist die Bedingugn sichergestellt. Gleichermaßen kann auch überprüft werden, ob das Programm Aktivitäten verwendet, die der Ausführende nicht freigegeben hat. Da diese statischen Überprüfungen zur Kompilierzeit unter anderem durch Schleifen oder bedingte Sprünge erschwert beziehungsweise unmöglich werden, muss die Einhaltung der nicht statisch prüfbaren Berechtigungen zur Laufzeit sichergestellt werden. Anders als bei anderen Usage Control Implementierungen wie beispielsweise IND^{2}UCE wird dazu jedoch keine zusätzlich im Hintergrund ausgeführte Software benötigt. Die dynamischen Berechtigungsprüfungen werden vom D°-Compiler direkt in das Kompilat für die jeweilige Zielsprache integriert oder an die Laufzeitumgebung delegiert. Beispielsweise kann die Pfadbeschränkung der Berechtigung „Es dürfen nur Dateien aus Pfad X mit einer maximalen Größe von Y gelesen werden“ an das Dateisystem delegiert werden, wohingegen die Prüfung der Dateigröße von gängigen Dateisystemen nicht unterstützt wird und somit in der Zielsprache erfolgen muss.

3.2 Entwicklung

Der *Code* für eine einzelne Applikation ist eine Instanz der Grammatik, welche die Definitionen von Aktivitäten, Datentypen und Polices verwendet. Dieser Code ist die Eingabe für die *Processing Pipeline*, welche aus mehreren Schritten besteht und am Ende eine ausführbare *Data App* erzeugt.

Während der Code die einzelnen Stufen der Processing Pipeline durchläuft werden verschiedene Aktionen durchgeführt. Nach dem Lexen und Parsen des Codes werden einige Prüfungen auf dem abstrakten Syntaxbaum des Codes ausgeführt. Dazu gehören zum Beispiel die Prüfungen, ob die im Code verwendeten Aktivitäten und Typen auch existieren und ob bei den Aktivitätsaufrufen die übergebenen Argumente vom richtigen Typ sind. Nach der oben beschriebenen statischen Analyse der Einhaltung von Policies bildet der *(Cross-) Compiler* den letzten Schritt der Processing Pipeline. Der aktuell entwickelte Cross-Compiler führt eine Übersetzung

in Java-Code durch und erlaubt hierdurch die Verwendung von D° auf einer Vielzahl von Geräten und Geräteklassen. Durch den hohen Abstraktionsgrad von D° entstehen bei der Übersetzung in eine andere Programmiersprache komplexe Applikationen in der Zielsprache, welche anschließend in ausführbaren Code übsersetzt werden.

Eine direkte Kompilierung in ausführbaren Maschinencode oder andere Programmiersprachen ist möglich, wird zum aktuellen Zeitpunkt aber nicht entwickelt.

Zusätzlich zu einer ausführbaren Applikation wird durch die Processing Pipeline ein Zertifikat erstellt, welches diverse Informationen über die Applikation enthält. Neben Informationen, welche zur Integritätsprüfung der ausführbaren Datei dienen, wird beschrieben, welche Teile der Nutzungsbedingungen bereits überprüft wurden und welche zur Laufzeit durchgesetzt werden müssen.

Das Gesamtpaket aus ausführbarer Applikation und Zertifikatsinformation wird im Kontext von D° als *Data App* bezeichnet.

3.3 Ausführung

Zur Ausführung der Data Apps wird eine spezielle *Runtime Environment* benötigt, welche die Verwaltung der Data Apps durchführt und Laufzeitpolicies durchsetzt. Ein *Permission Center* liest die Informationen aus den Zertifikaten der Data Apps und entscheidet, ob bestimmte Aktionen bei einer gegebenen Policy ausgeführt werden dürfen. An dieser Stelle werden auch die dynamischen Teile der Policy durchgesetzt, welche nicht zur Übersetzungszeit überprüft werden konnten. Über eine *Management API* ist es dem Anwender möglich Data Apps zu steuern und Einstellungen anzupassen.

Dabei ist es möglich, dass einzelne Aktivitäten durch unterschiedliche Technologien und Schnittstellen (beispielsweise command-line Interface, REST-Service) realisiert und erreichbar sind. In diesen Fällen wird durch die Ausführungsumgebung eine Delegation durchgeführt, welche von der Data App ausgelöst wird.

3.3.1 Remote Processing

Da es in bestimmten Szenarien erforderlich sein kann, dass die Daten von Partnern verarbeitet werden und gleichzeitig im eigenen Unternehmen verbleiben, enthält D° eine Reihe von Funktionen, die genau diese Problematik adressieren und unter dem Begriff des Remote Processings zusammengefasst sind.

Abbildung 2 zeigt schematisch die beiden Akteure des Remote Processings und welche Komponenten von D° beim Datenbesitzer vorhanden sein müssen, um empfangene Data Apps zu übersetzen und auszuführen.

Da jede Data App in einer eigenen Sandbox betrieben wird, findet an dieser Stelle eine Isolation vom ausführenden System statt, welche das System und die gespeicherten Daten vor unerwünschten Zugriffen schützt. Es ist die Aufgabe des Datenbesitzers, der Data App die notwendigen Daten zur Verfügung zu stellen und dabei definierte Schnittstellen (beispielsweise festgelegte Datenbankschemata) einzuhalten. Werden die Daten in die Sandbox der Data App dupliziert, ist darüber hinaus auch die Integrität der Originaldaten gewahrt, unabhängig von den ausgeführten Aktivitäten der Data App.

Die Kommunikation zwischen Datennutzer und Datenbesitzer findet über definierte Schnittstellen statt, wodurch dieser Vorgang auch automatisiert stattfinden kann.

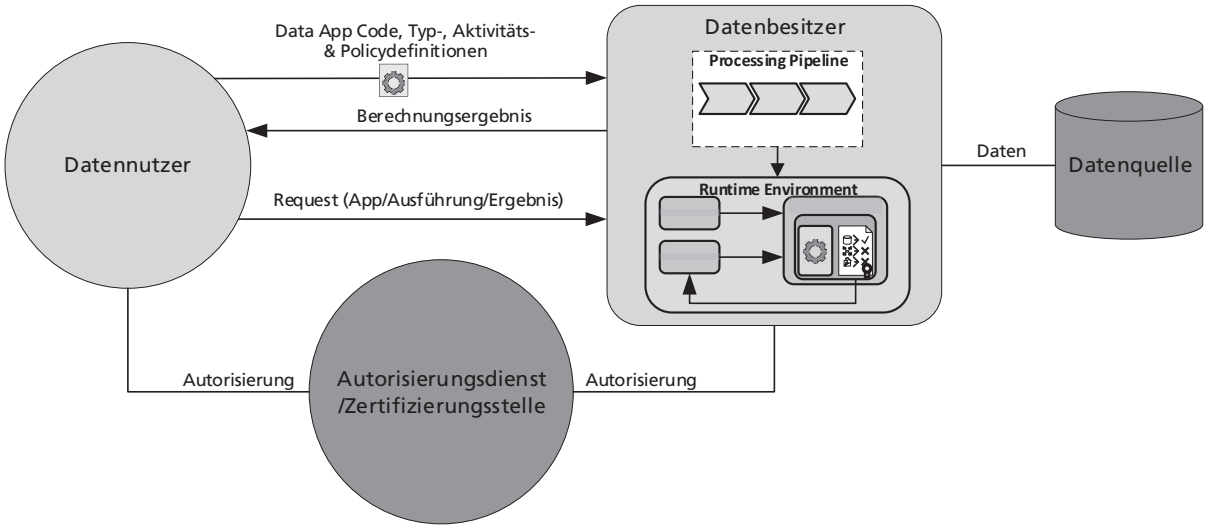

Abb. 2: Schematische Darstellung der Remote Processing Funktionalität von D°

4 Beispiel

Nachdem die verschiedenen Aspekte und Konzepte von D° vorgestellt wurden, soll ein Beispiel die Nutzung veranschaulichen. Dabei wird zunächst das Szenario umrissen:

Ein Webshop verwendet D° zur Abwicklung von Bestellungen. Dabei spezifiziert die Policy der Bestellabwicklung, dass eine Bestellung nur dann versendet werden darf, wenn während der Abwicklung die Versandadresse validiert wurde. Diese Eigenschaft kann während der Übersetzung statisch geprüft und sichergestellt werden, indem alle Ausführungspfade auf die Existens einer entsprechenden Aktivität hin überprüft werden.

```
...
PositiveAnzahlValidator := $ScriptValidator(
    @set["script", "anzahl >= 0"]
)
WarenkorbEintrag := $Composite(
    @validate[$PositiveAnzahlValidator],
    @attribute["artikel", $Artikel],
    @attribute["anzahl", $Integer]
)
Bestellung := $Composite(
    @attribute["kunde", $Kunde],
    @attribute["rechnungsadresse", $Adresse],
    @attribute["lieferadresse", $Adresse],
    @attribute["warenkorb", WarenkorbEintrag[]]
)
```

Abb. 3: Definition von Datentypen in D°

Außerdem muss der Webshop sicherstellen, dass seine Prozesse die Anforderungen der DSGVO erfüllen, weswegen nur an Kunden versendet werden darf, welche zuvor der Datennutzung zu-

```
...
Fakturierung := Activity(
    @input["rechnungsadresse", $Postadresse],
    @requires["rechnungsadresse", "validated"],
    @input["warenkorb", $WarenkorbEintrag[]],
    @requires["warenkorb", "available"]
)
BestellungBearbeiten := $Activity(
    @input["bestellung", $Bestellung]
    begin
        [verfuegbarerWarenkorb, nichtVerfuegbarerWarenkorb] =
           LagerbestandPruefen[bestellung.warenkorb];
        [solvent] = Bonitaetspruefung[bestellung.kunde];
        [lieferadresseGueltig] = AdressePrüfen[bestellung.
           lieferadresse];
        if(solvent && lieferadresseGueltig) begin
            Versand[bestellung.lieferadresse, verfuegbarerWarenkorb];
            Fakturierung[bestellung.rechnungsadresse,
               verfuegbarerWarenkorb];
        end
    end
)
```

Abb. 4: Definition von Aktivitäten in D°

gestimmt haben. Da dies für jeden Datensatz individuell ist, kann es erst zur Laufzeit entschieden werden.

Abbildung 3 zeigt die Definition der Datentypen `Bestellung`, `WarenkorbEintrag` und den Validator `PositiveAnzahlValidator`, welcher sicherstellt, dass die Artikel im Warenkorb einen positiven Lagerbestand haben. Dabei sind die folgenden syntaktischen Merkmale erkennbar. Die Zeichenkette `:=` wird in D° als Klon-Operator verwendet und weist dem Bezeichner auf der linken Seite eine Kopie des Wertes auf der rechten Seite zu. Für die Referenzierung von bereits existierenden Elementen wird innerhalb von D° das Zeichen `$` als Referenzoperator verwendet. Um beim Klonen von Elementen Anpassungen vornehmen zu können ist es notwendig, auf Methoden zur Manipulation zugreifen zu können. Hierfür wird das `@`-Zeichen als Schalter verwendet, welcher dafür sorgt, dass der nachfolgende Identifier als Methodenzugriff interpretiert wird.

In Abbildung 4 werden zwei Aktivitäten definiert, welche die Datentypen aus Abbildung 3 verwenden und den zuvor beschriebenen Workflow in Teilen abbilden. Dabei handelt es sich bei der Aktivität `BestellungBearbeiten` um eine zusammengesetzte Aktivität, welche keine Implementation in einer anderen Sprache besitzt, wogegen die Aktivität `Fakturierung` atomar ist und die Implementierung in einer anderen Sprache vorliegen muss.

Die gezeigten Beispiele zeigen den aktuellen Stand der Entwicklung und können sich zukünftig noch ändern. Es werden verschiedene Konzepte getestet und auf Basis von Fallstudien evaluiert und ggf. iterativ angepasst.

5 Resümee

In dieser Arbeit wurde der Aufbau der domänenspezifischen Programmiersprache D°, welche sich aktuell in der Entwicklung befindet, vorgestellt. Darüber hinaus wurden wichtige Konzepte vorgestellt, welche Alleinstellungsmerkmale der Sprache sind. Außerdem wurde ein Nutzungsbeispiel auf Basis des aktuellen Entwicklungsstandes präsentiert.

Die Grundlage für diese Konzepte ist die erweiterbare Struktur von D°, welche es den Nutzern erlaubt, die Sprache gemäß den eigenen Anforderungen zu erweitern. Es wurde der Aufbau des erweiterbaren Typsystems aufgezeigt und die eingebauten Mechanismen zur Definition und Durchsetzung von Nutzungsbedingungen vorgestellt. Außerdem wurden die Funktionen der Sprache vorgestellt, welche eine automatisierte und entfernte Übersetzung, Ausführung sowie Rücksendung der Ergebnisse erlauben.

Danksagung
Teile dieser Ausarbeitung entstanden im Leistungszentrum Logistik und IT in Dortmund.

Literatur

[AJRO12] S. Ackermann, V. Jovanovic, T. Rompf, M. Odersky: Jet: An Embedded DSL for High Performance Big Data Processing. In *International Workshop on End-to-end Management of Big Data (BigData 2012)*, 2012.

[CZ12] D. Chen and H. Zhao: Data security and privacy protection issues in cloud computing. In *2012 International Conference on Computer Science and Electronics Engineering*, volume 1, pages 647–651, March 2012.

[Dor14] M. Dorner: Big data und "dateneigentum". *Computer Und Recht*, 30(9):617–628, 2014.

[FG05] K. Fisher, R. Gruber: Pads: a domain-specific language for processing ad hoc data. In *ACM Sigplan Notices*, volume 40, pages 295–304. ACM, 2005.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud: The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[HHKW77] M. Hammer, W.G. Howe, V.J. Kruskal, I. Wladawsky: A very high level programming language for data processing applications. *Communications of the ACM*, 20(11):832–840, 1977.

[Jlu16] Dennis R. Jlussi. Industrie 4.0 und dateneigentum. *Industrie 4.0 in Eckpunkten – Ein interdisziplinärer Querschnitt*, 2016.

[LKY14] J. Lee, H.-A. Kao, S. Yang: Service innovation and smart analytics for industry 4.0 and big data environment. *Procedia CIRP*, 16:3 – 8, 2014. Product Services Systems and Value Creation. Proceedings of the 6th CIRP Conference on Industrial Product-Service Systems.

[MPB03] M. Casassa Mont, S. Pearson, P. Bramhall: Towards accountable management of identity and privacy: Sticky policies and enforceable tracing services. In *Database and Expert Systems Applications, 2003. Proceedings. 14th International Workshop on*, pages 377–382. IEEE, 2003.

[OCM+16] A. Otto, J. Cirullies, S. Menz, et al: Industrial data space-digitale souveränität über daten. *München: Fraunhofer-Gesellschaft zur Förderung der angewandten Forschung eV*, 2016.

[Oea18] B. Otto et al: Ids reference architecture model – industrial data space version 2.0. http://s.fhg.de/pmL, zugegriffen am 07.06.18, 2018.

[OJS+16] B. Otto, J. Jürjens, J. Schon, S. Auer, N. Menz, S. Wenzel, J. Cirullies: Industrial data space – whitepaper. http://s.fhg.de/eGW, 2016.

[OLA+17] B. Otto, S. Lohmann, S. Auer, G. Brost, J. Cirullies, A. Eitel, T. Ernst, C. Haas, M. Huber, C. Jung, J. Jürjens, C. Lange, C. Mader, N. Menz, R. Nagel, H. Pettenpohl, J. Pullmann, C. Quix, J. Schon, D. Schulz, J. Schütte, M. Spiekermann, S. Wenzel: Reference architecture model for the industrial data space. http://s.fhg.de/L7q, zugegriffen am 07.06.18, 2017.

[ORS+08] C. Olston, B. Reed, U. Srivastava, R. Kumar, A. Tomkins: Pig latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[Ott15] B. Otto: Quality and value of the data resource in large enterprises. *Information Systems Management*, 32(3):234–251, 2015.

[PCM11] S. Pearson, M. Casassa-Mont: Sticky policies: an approach for managing privacy across multiple parties. *Computer*, 44(9):60–68, 2011.

[PGB11] Z. N.J. Peterson, M. Gondree, R. Beverly: A position paper on data sovereignty: The importance of geolocating data in the cloud. 2011.

[SBW18] J. Schütte, G. Brost, S. Wessel: Datensouveränität im internet der dinge – der trusted connector im industrial data space. http://s.fhg.de/MHy, zugegriffen am 07.06.18, 2018.

[SKJH16] M. Steinebach, E. Krempel, C. Jung, M. Hoffmann: Datenschutz und datenanalyse. *Datenschutz und Datensicherheit-DuD*, 40(7):440–445, 2016.

[Tan08] Q. Tang: On using encryption techniques to enhance sticky policies enforcement. 2008.

[TS12] S. Trabelsi, J. Sendor: Sticky policies for data control in the cloud. In *Privacy, Security and Trust (PST), 2012 Tenth Annual International Conference on*, pages 75–80. IEEE, 2012.

[VDKV+00] A.Van Deursen, P. Klint, J. Visser et al: Domain-specific languages: An annotated bibliography. *Sigplan Notices*, 35(6):26–36, 2000.