Dokumentation MOSAIK - FAU

Dokumentation und Tutorial zum Umgang mit ldfu und BOLD am Beispiel eines einfachen Transportszenarios

von Daniel Schraudner, Sebastian Schmid, Andreas Harth

Inhalte der Dokumentation:

- Crashkurs Begrifflichkeiten (bspw. Agent, Affordanz, Stigmergie)
- Crashkurs RDF (im Kontext unserer Infrastruktur)
- HowTo ldfu
- HowTo BOLD
- HowTo Datenmodell und arena-Ontologie
- HowTo Schnittstellen zwischen Agenten und Artefakten (ShEx, OpenAPI, etc.)
- Definition Transportszenario (aka was soll umgesetzt werden)
    - Usecase: Was wollen wir damit machen?
    - Umsetzung: Wie wird es am Ende gemacht?

Crashkurs Begrifflichkeiten

Software-Agent
Proaktive, zustandslose Komponenten (im Ggs zu Artefakten) in unserem System, welche selbstständig Entscheidungen treffen. Implementieren einen Wahrnehmungs-Nachdenken-Aktions-Zyklus zum Beeinflussen ihrer Umgebung, basierend auf definierten Bedingungen und Aktionsregeln. Ziele des Gesamtsystems ergeben sich aus Interaktionen der einzelnen Agenten und deren Ziele.
Die Wahrnehmung und Aktionen des Agenten werden durch HTTP-Requests umgesetzt. Agenten bestehen ausschließlich aus Software, welche auf beliebigen System laufen können.

Artefakt
Reaktive Komponenten (im Ggs. zu Agenten), welche Teil der Umgebung sind. Stellen vier verschiedene Möglichkeiten bereit mit ihnen zu interagieren: das Auslesen und Manipulieren von Eigenschaften (Properties), das Ausführen von längerfristigen Aktionen (über Tasks), das Bereitstellen von Events und Blackboards. Interaktionen mit den Artefakten werden über HTTP Request realisiert.
Eine interne Logik der Artefakte definiert die Reaktion auf Interaktion. Das Verhalten an sich ist für Agenten eine Black Box, das Resultat des Verhaltens jedoch wieder über Properties, Tasks und Events beobachtbar. Änderungen des Zustands können nicht nur durch Agenten, sondern auch durch die physikalische Umwelt angeregt werden (Bsp.: Fehlerzustand an Maschine, mechanische Überlastung, Sensorfehler, Kabelbruch).

Umgebung
Graph bestehend aus Artefakten und Informationsressourcen, welche über Hypermedia verbunden sind. Agenten können durch diesen Graph navigieren, indem sie Hyperlinks folgen. Agenten können den Graph bspw. manipulieren, indem sie neue Kanten zwischen Artefakten und Informationsressourcen erstellen oder den Wert von Properties verändern.

Informationsressource
RDF-Dokument welches als Web Ressource verfügbar gemacht wird und von Agenten beliebig gelesen und manipuliert werden kann.

Stigmergie
Kommunikationsparadigma, bei welchem Agenten nicht direkt miteinander, sondern nur indirekt über ihre gemeinsame Umgebung interagieren und kommunizieren. In unserem System können Agenten kommunizieren, in dem sie Artefakte manipulieren. Das bedeutet Properties und Tasks manipulieren, das Blackboard des Artefakts oder Informationsressourcen in der Umgebung.

Affordanz
In unserem System sind Affordanzen die Interaktionsmöglichkeiten von Artefakten, um mit ihnen über Properties, Tasks und Events zu interagieren. Agenten müssen dieses Schema erkennen können. Aktionen, als Untermenge der Affordanzen, können zusätzlich mit Vor- und Nachbedingungen ausgezeichnet werden.

Property
Teil des Zustands eines Artefakts, der entweder gelesen, geschrieben oder beides werden kann.

Action
Zustandsänderungen, welche über einen längeren Zeitraum durchgeführt werden müssen und nicht instant erfolgten. Werden von einem Artefakt genau dann durchgeführt, wenn sie einen entsprechenden Task erhalten.

Task
Ressourcen, welche auf einem Artefakt von Agenten angelegt werden, um die Ausführung einer Aktion anzufragen und die Parameter hierfür zu spezifizieren. Tasks werden in der von den Agenten vorgegebenen Reihenfolge so bald wie möglich von Artefakten abgearbeitet.

Event
Nicht-persistente Information von Artefakten über eine Zustandsänderung, welche von Agenten wahrgenommen werden kann.

Crashkurs RDF

Resource Description Framework (RDF) bezeichnet einen W3C Standard zur Beschreibung und Formulierung von Aussagen über beliebige Dinge sog. Ressourcen. Diese folgen dabei dem Schema (sog. Tripel):

	Subjekt	Prädikat	Objekt
Bsp:	http://example.org/lamp	rdf:type	:Lamp

Klartext: “Die Ressource hinter der URI http://example.org/lamp hat den RDF-Typ “Lampe””.
Die Verknüpfung von verschiedenen Tripeln zu einander (bspw. kann das Objekt eines Tripels wiederum das Subjekt eines anderen sein), ergibt in Summe ein graphbasiertes Datenmodell. Ein Agent, welcher dieses Tripel auswertet kann nun bspw. Rückschlüsse auf die Ressource mit der definierten Eigenschaften des Typs “Lampe” ziehen.

Artefakte stellen unter den unten genannten Endpunkten immer RDF-Dokumente zur Verfügung, welche durch die Agenten interpretiert und manipuliert werden. Der Standard hierbei ist die Turtle-Serialisierung (.ttl).
Informationsressourcen sind immer RDF-Dokumente in ttl-Serialisierung. Sämtliche vorkommende URIs können dereferenziert werden und geben für diese Ressource relevante Tripel zurück.

Howto ldfu

ldfu ist ein Programm zum Abrufen, Verarbeiten und Verändern von Linked Data basierend auf logischen Regeln und Produktionsregeln in n3. Wir verwenden es um die Agenten zu implementieren.
Step by step:

- ldfu Version 0.9.12 herunterladen https://github.com/aharth/linked-data-fu.git
- ./bin/ldfu.sh
    - Binaries sind bereits vorhanden und bereitgestellt. Kein Rekompilieren notwendig
- Startpunkt(e) im Hypermedia-Graphen über -i definieren
    - Der Startpunkt ist eine URI
    - Bei einer Simulation mit BOLD ist der Startpunkt einer der selbstdefinierten Endpunkte
    - Bei einer Verwendung mit einer echten Umgebung ist der Startpunkt eine Informationsressource, welche möglichst viele Links (auch indirekt) zu anderen Informationsressourcen / Artefakten besitzt.
- Wahrnehmungsprogramm über -p definieren
    - geschrieben in n3.
    - Definiert eine oder mehrere n3-Regeln für Agenten um schrittweise zu Selektieren, welchen Links gefolgt werden soll um die Umgebung wahrzunehmen
    - Die Selektion kann über Basic Graph Patterns definiert werden. D.h. wenn definierte RDF-Tripel (mit möglichen Variablen) wahrgenommen werden, wird einem Teil der Variablen gefolgt und deren Ressource ebenfalls in die Wahrnehmung mit einbezogen.
    - Hängt von der Struktur des Hypermedia-Graphen ab
    - Bsp: { ?s ldp:contains ?o } => { [] http:mthd httpm:GET ; http:requestUri ?o .} .
- Aktionsprogramm über -p definieren
    - geschrieben in n3.
    - Definiert eine oder mehrere n3-Regeln für Agenten um Interaktionen mit Artefakten unter gewissen Bedingungen auszuführen
    - Die Interaktion kann über Basic Graph Patterns definiert werden. D.h. wenn definierte RDF-Tripel (mit möglichen Variablen) wahrgenommen werden, wird eine Interaktion mit der RDF-Ressource ausgeführt. Die Ressource ergibt sich dabei aus einer Variable, die restlichen Variablen können dabei Paramter für die Interaktion sein.
    - Bsp:

```
{
        ?lamp  rdf:type :Lamp;
               arena:hasPropertyContainer ?container .

        ?container ldp:contains ?onOffProperty .

        ?onOffProperty rdf:type :OnOffProperty ;
               rdf:value false .
```

```
} => {
        []      http:mthd httpm:PUT ;
                http:requestUri ?onOffProperty ;
                http:body {
                        ?onOffProperty rdf:type :OnOffProperty ;
                                rdf:value true .
                } .
} .
```

- Kommentar zum obigen n3-Aktionsprogramm:
  - Wenn eine Ressource vom Typ ‘Lamp’ wahrgenommen wird und diese eine Eigenschaft vom Typ ‘OnOffProperty’ hat, welche den Wert ‘false’ besitzt, schicke einen HTTP PUT Request an die Adresse der Eigenschaft mit dem Wert ‘true’.
  - D.h., wenn der Agent eine Lampe (ein Artefakt) wahrnimmt, welche derzeit den Status ‘aus’ hat, so schickt der Agent einen HTTP PUT Request um die Lampe einzuschalten (vorausgesetzt die Interaktion mit dem Artefakt wurde so definiert).

Damit die Agenten dauerhaft in einer Schleife laufen (und so ihre Wahrnehmung und Aktion erneut durchgeführt wird), muss ein Argument -n an ldfu übergeben werden. Es ist sinnvoll, dabei eine Verzögerung zwischen den einzelnen Ausführungen in Millisekunden anzugeben (hinter dem Parameter -n eine Zahl in ms angeben, bspw. 500 oder 1000). Sie sollte so groß gewählt sein, dass die Systeme nicht durch zu viele HTTP Requests überlastet werden und die Artefakte genügend Zeit haben um darauf zu reagieren, aber auch klein genug, dass der Agent keine Informationen aus der Umgebung verpasst, die für ihn von interesse wären.

Bsp. für einen kompletten (fiktiven) ldfu Aufruf:
./bin/ldfu.sh -i http://example.org/startpoint -p perceptionRules.n3 -p actionRules.n3 -n 500

ldfu gibt während der Ausführung alle gesendeten HTTP Requests und deren Resultat auf der Konsole aus.

Howto BOLD

BOLD ist eine Simulationsumgebung die mehrere RDF-Graphen verwalten kann. Für jeden dieser Graphen existiert ein HTTP Endpunkt (unter der URI des Named Graphs), über welchen Agenten das RDF lesen und manipulieren können. Es können Simulationsregeln global für alle Graphen definiert werden, welche über SPARQL Insert / Delete Requests umgesetzt werden. BOLD wendet diese Regeln auf das gesamte RDF in festgelegten Zeitschritten an. Die Simulation kann über das REST Interface gestartet und gestoppt werden; ein Dump des gesamten RDFs über alle Zeitschritte kann erstellt werden.

BOLD besteht aus folgenden Konfigurationsdateien auf welche die Config-Datei verweist:

- initialization Dataset
    - eine .trig Datei, bei jeder Named Graph einen Endpunkt repräsentiert und die Tripel in diesem Named Graph unter dem Endpunkt zur Verfügung gestellt werden. Diese Daten werden am Anfang in die Simulation geladen
- initialization SPARQL Update
    - SPARQL Requests welche noch vor Start der Simulation auf die Daten ausgeführt werden, bspw. um Zufallszahlen zu erzeugen.
- runtime SPARQL Updates
    - SPARQL Requests welche während der Simulation ausgeführt werden und die eigentlichen Simulationsregeln darstellen. Werden in einer Schleife bis zum Ende der Simulation ausgeführt.
    - Bestehen idR aus SPARQL Insert / Delete Requests
    - im Where-Teil müssen keine Named Graphs verwendet werden, sondern nur bei Insert und Delete (um die entsprechenden Graphen anzusprechen)
- runtime SPARQL Queries
    - SPARQL Queries welche während der Simulation eine definierte Ausgabe pro Zeitschritt erzeugen und auf die Konsole ausgeben. Werden in einer Schleife bis zum Ende der Simulation ausgeführt.
- RDF dumps
    - Nach dem Ende der Simulation werden in einer “Replay”-Phase die Schritte der Simulation noch einmal schnell durchgegangen. Hierbei werden die RDF dumps erzeugt und gespeichert (Speicherort wird in der Config-Datei vorher definiert).

Step by step:

- https://github.com/wintechis/ldp-sail installieren
- https://github.com/wintechis/bold-server-ldp.git herunterladen
- gradle install (Gradle Version 7 funktioniert im Zweifel)
- Config-Datei anlegen
    - Enthält den Speicherort des Datasets und der Queries (siehe oben)
    - Dump-Funktion kann eingeschaltet werden und Speicherort festgelegt werden
    - Port wird festgelegt
- ./bin/bold-server <Config-Datei>, um den Server zu starten

- Um die Simulation zu starten, muss an den Endpunkt /sim RDF per PUT gesendet werden, das den Startzeitpunkt der Simulation, die Anzahl der durchzuführenden Schritte, sowie die Dauer der einzelnen Schritte spezifiziert (siehe run.sh)
    - Bsp für RDF welches an /sim gesendet wird:
      <sim> :initialTime "2020-05-21T08:00:00Z"^^xsd:dateTime ;
      :timeslotDuration 60000 ; # in ms (corresponds to simulated time, not real execution time)
      :iterations 1440 .
- Um die Simulation vorzeitig zu beenden, kann an den /sim Endpunkt erneut das im Schritt vorher beschriebene RDF per PUT geschickt werden
- Die Simulation endet nach den vorgegebenen Schritten automatisch und eine evtl. Replay-Phase wird durchgeführt

Beispiel für Config-Dateien und BOLD-Aufruf:
Config-Datei "config.properties":

```
# initialization
bold.init.dataset = data/*.trig
bold.init.update = query/init-sim-*.rq

# runtime configuration
bold.runtime.update = query/update-sim-*.rq
bold.runtime.query = query/sim-*.rq

# replay configuration
# %d: iteration number
bold.replay.dump = dump/dataset-%d.trig

# Server configuration
# default: 8080
bold.server.httpPort = 8080
```

Dataset "lamp.trig":

```
<http://example.org/lamp> {
        <http://example.org/lamp> rdf:type :Lamp ;
                arena:hasPropertyContainer <./properties> .
}

<http://example.org/lamp/properties> {
        <http://example.org/lamp/properties> rdf:type arena:PropertyContainer ;
                ldp:contains <./onOff>, </illuminance> .
}

<http://example.org/lamp/properties/illuminance> {
        <http://example.org/lamp/properties/illuminance> rdf:type IlluminanceProperty ;
                rdf:value 0 .
}
```

Kommentar:
Es wird eine Lampe definiert mit dem Endpunkt …/lamp und einem zugehörigen PropertyContainer, welcher unter …/lamp/properties erreichbar ist. In diesem liegen dann die Properties und die zugehörigen Endpunkte zu jenen, bspw. …/lamp/properties/illuminance. Es werden somit ausschließlich die Endpunkte und das dort verfügbare RDF definiert.

initalization / runtime SPARQL Update:

```
DELETE {
        GRAPH ?illuminance {
                ?illuminance rdf:value ?value .
        }
} INSERT {
        GRAPH ?illuminance {
                ?illuminance rdf:value ?newValue .
        }
} WHERE {
        ?lamp a :Lamp ;
                arena:hasPropertyContainer ?container .

        ?container ldp:contains ?illuminance .

        ?illuminance rdf:type :IlluminanceProperty ;
                rdf:value ?value .

        ( ?value 1 ) math:sum ?newValue .
}
```

Kommentar:
Wähle eine RDF Ressource vom Lamp (also eine Lampe) und ermittle ihre Helligkeit über das Property IlluminanceProperty. Nehme den derzeitigen variablen Wert ?value, addiere auf diesen 1 drauf und speichere es in der Variable ?newValue. Anschließend, lösche das Tripel dieses Properties mit dem Wert ?value und schreibe stattdessen das Tripel mit dem Wert ?newValue.
Als initalization SPARQL Update wird das obige Programm nur einmal am Anfang ausgeführt (und somit die Helligkeit nur einmal erhöht), als runtime SPARQL Update würde die Helligkeit mit jedem Zeitschritt um 1 erhöht werden.

RDF dumps

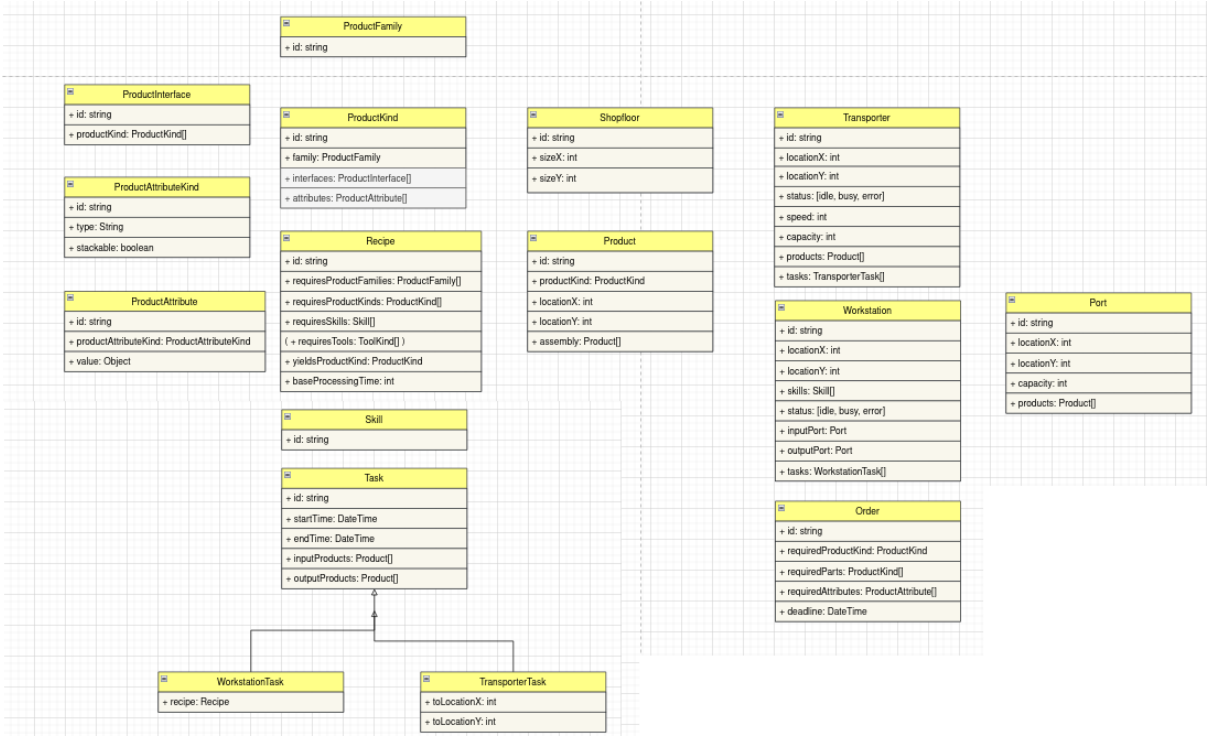
```
<http://example.org/lamp> {
        <http://example.org/lamp> rdf:type :Lamp ;
                arena:hasPropertyContainer <./properties> .
}

<http://example.org/lamp/properties> {
        <http://example.org/lamp/properties> rdf:type arena:PropertyContainer ;
                ldp:contains <./onOff>, </illuminance> .
}
```

<http://example.org/lamp/properties/illuminance> {
<http://example.org/lamp/properties/illuminance> rdf:type IlluminanceProperty ;
rdf:value 42 .
}

Kommentar:
Der Dump sieht im Prinzip genauso aus, wie das Initialization-Dataset, spiegelt aber natürlich die Änderungen durch die Simulation und die Agenten wider. Hier: illuminance hat einen konkreten Wert von 42.

HowTo Datenmodell und arena-Ontologie



- Transporter und Workstations
  - stellen Artefakte dar
  - haben Property-Action-Events-Interface
  - Workstations können durch Actions Recipes ausführen
  - Workstations können Ports haben
  - Transporter können durch Actions Produkte auf dem Shopfloor bewegen
  - Transporter wissen immer, welche Produkte sie gerade geladen haben
- Recipes
  - haben Inputs und Outputs an Produkten
  - beschreiben einen Arbeitsschritt
  - setzen einen Skill voraus, damit sie ausgeführt werden können
  - können auf Workstations ausgeführt werden, welche den passenden Skill dazu haben
  - Recipes können auf Workstations nur durch Actions ausgeführt werden

- Products
    - alle Teile, d.h. alle Ressource, Zwischenprodukte und das Endprodukt
    - sind rein passiv
    - Recipes beschreiben, wie Products an Workstations transformiert werden können (in andere Products)
    - können verschiedene Eigenschaften haben, welche über ProductFamily, Attribute und Kind genauer definiert werden
- Skills
    - Sind im Prinzip nur ein Name, um die Verbindung zwischen Recipes und Workstations herzustellen
- Order
    - kommt von Außen in das System
    - beschreibt das Endprodukt, welches gefertigt werden soll (das “Ziel”), allerdings nicht den Weg dahin
    - hat eine Deadline
- Task
    - Wird verwendet, um einem Artefakt mitzuteilen, dass es eine bestimmte Action ausführen soll
    - Unterscheidung zwischen WorkstationTask und TransporterTask
    - WorkstationTasks definieren ein Recipe das ausgeführt werden soll
    - TransporterTasks definieren Zielkoordinaten und welche Produkte dort abgelegt und/oder aufgeladen werden sollen
- Shopfloor
    - Stellt die Grenzen des Shopfloors (Koordinationsystem)
- Ports
    - Workstations können Input- und Output-Ports haben über die sie Produkte konsumieren und erzeugen
    - Die Ports wissen immer, welche Produkte gerade in ihnen liegen

Die ARENA-Ontologie unter https://solid.ti.rw.fau.de/public/ns/arena ist eine direkte Umsetzung dieser Klassen.

HowTo Schnittstellen zwischen Agenten und Artefakten

Artefakte stellen ein Linked Data REST Interface zur Verfügung. Es gibt folgende Endpunkte (dem Aufbau nach ähnlich wie Datasets in BOLD):

- /<artefakt>/: der Zustand des Artefakts kann über einen HTTP GET Request abgefragt werden. In der Antwort sind die Werte aller Properties, der momentan ausgeführte Task sowie Links zu allen Events enthalten. Außerdem gibt es Links zum PropertyContainer, TaskContainer, EventContainer und InformationContainer (Blackboard)
- /<artefakt>/properties/: Ein LDP-BasicContainer der alle Properties des Artefakts enthält (nur GET)

- /<artefakt>/tasks/: Ein LDP-Container, der eine Priority Queue implementiert und alle Tasks des Artefakts enthält (nur GET, POST)
- /<artefakt>/events/: Ein LDP-Container, der alle Events enthält deren Zeitstempel noch nicht abgelaufen ist (nur GET)
- /<artefakt>/information/: Ein LDP-BasicContainer der alle zum Artefakt gehörigen Informationsressourcen (können beliebige Ressource - auch weitere Container – sein) enthält (nur GET, POST)
- /<artefakt>/properties/<propertyname>: Enthält den Wert des property und mit welchem Prädikat es im Artefakt-Endpunkt angezeigt wird (nur GET, PUT (wenn schreibbar))
- /<artefakt>/tasks/<taskid>: Enthält einen Task (nur GET, PUT, DELETE)
- /<artefakt>/events/<eventid>: Enthält ein Event (nur GET)

Beispiel 1: Agent will Licht an einem Transporter anschalten sofern es aus ist
Aus der Sicht des Agenten beschrieben:

- Beginne beim Einstiegspunkt in den Hypermedia-Graphen
- Suche alle Artefakte, die Transporter sind
- Suche alle Transporter, welche in ihren PropertyContainer eine Property haben, die auf lt. Ontologie ein Lampen-Property ist
- Nimm alle diese Lampen-Properties, welche den Wert “false” haben (also aus sind)
- Sende an direkt an die Property-Ressource einen HTTP PUT Request mit dem Wert “true” (dh die Lampe soll eingeschaltet werden)

```
{
        ?lamp  rdf:type :Lamp;
                arena:hasPropertyContainer ?container .

        ?container ldp:contains ?onOffProperty .

        ?onOffProperty rdf:type :OnOffProperty ;
                rdf:value false .
} => {
        []      http:mthd httpm:PUT ;
                http:requestUri ?onOffProperty ;
                http:body {
                        ?onOffProperty rdf:type :OnOffProperty ;
                                rdf:value true .
                } .
} .
```

Beispiel 2: Agent will ein Mainmodule mit der vorher generierten URI unter Variable ?mainmodule herstellen. CPU und Mainboard wurden bereits produziert und liegen an einer Soldering Station. Das Rezept für ein Mainmodule ist noch unbekannt.

- Beginne beim Einstiegspunkt in den Hypermedia-Graphen
- Suche alle Rezepte um Mainmodules herzustellen
- Es wird bekannt, dass wir hierzu eine CPU, ein Mainboard und eine Workstation mit einem Soldering Skill brauchen
- suche alle Workstation mit einem Soldering Skill

- Nimm diese Soldering Stationen und diejenigen, welche bereits eine CPU und ein Mainboard im Input-Port liegen haben und platziere einen WorkstationTask (mit Link auf die spezifischen CPUs und Mainmodules im Input-Port, sowie auf ?mainmodule) im TaskContainer einer dieser Workstations

```
{
        ?recipe rdf:type arena:Recipe ;
                arena:requiresProductKinds arena:CPU, arena:Mainboard ;
                arena:yieldsProductKind arena:MainModule ;
                arena:requiresSkill ?skill .

        ?station rdf:type arena:Workstation;
                arena:skill ?skill;
                arena:inputPort ?inputPort,
                arena:tasks ?taskContainer.

        ?inputPort arena:products ?cpu, ?mainboard .

        ?cpu rdf:type arena:Product ;
                arena:productKind arena:cpu .

        ?mainboard rdf:type arena:Product ;
                arena:productKind arena:mainboard .

} => {
        []      http:mthd httpm:POST ;
                http:requestUri ?taskContainer ;
                http:body {
                        <> rdf:type arena:WorkstationTask;
                                arena:recipe ?recipe;
                                arena:input ?cpu, ?mainboard;
                                arena:output ?mainmodule .
                } .
} .
```

Beispiel 3: Ein Agent wartet auf das Event welches den Abschluss der Herstellung des Mainmodules signalisiert, um anschließend einen Transporter zu rufen.

- Beginne beim Einstiegspunkt in den Hypermedia-Graphen
- Crawle alle Artefakte nach verfügbaren Events
- Filtere nach Events zum Abschluss von Mainmodule-Herstellung
- Suche daraufhin alle freien Transporter
- Schicke einen Transporter zum Ort der Workstation per Task

```
{
        ?station rdf:type arena:Workstation;
                arena:events ?event ;
                arena:outputPort ?outputPort.

        ?outputPort arena:locationX ?locX;
                        arena:locationY ?locY .
```

```
        ?event rdf:type arena:ProductionFinishedEvent ;
                arena:output ?mainModule .

        ?mainModule rdf:type arena:Product ;
                arena:productKind arena:mainModule .

        ?transporter rdf:type arena:Transporter ;
                arena:status arena:idle ;
                arena:tasks ?taskContainer .

} => {
        []      http:mthd httpm:POST ;
                http:requestUri ?taskContainer ;
                http:body {
                        <> rdf:type arena:TransporterTask;
                                arena:toLocationX ?locX ;
                                arena:toLocationY ?locY ;
                                arena:input ?mainModule .
                } .
} .
```

ShEx:
definiert wie die Form des RDFs für die einzelnen Ressourcen auszusehen hat

OpenAPI:
definiert welche Endpunkte existieren, welche Shapes bei GET zurückkommen / bei PUT & POST im Body stehen dürfen

Definition Transportszenario

Als Tutorial zum Verständnis der MOSAIK Laufzeitumgebung erstellen wir ein grundlegendes Transportszenario mit allen Komponenten.

Anwendungsbeschreibung

Ein quadratischer Shopfloor mit 7 x 7 Feldern besitzt 4 Stationen mit je einer von vier unterschiedlichen Farben grün, rot, blau und gelb. Eine Station nimmt nur Produkte von der eigenen Farbe an. Stationen produzieren zufällig ein Produkt einer anderen Farbe, wenn ihr Output-Port leer ist.
Transporter sollen farbige Produkte zu der korrekten Stationen der gleichen Farbe bringen. Agenten verwalten diese Tasks. Die Agenten der Transporter sollen hierbei nur das Feld, auf welchem der Transporter steht, und die 8 benachbarten Felder um den Transporter herum auswerten können. Die Agenten besitzen weiterhin keinen eigenen Speicher, sondern schreiben alle ihre Informationen in die Umgebung der Shopfloor Felder, so dass lokal alle Informationen miteinander geteilt werden

Transporter, Shopfloor und Stationen sollen in diesem Szenario Artefakte sein. Weiterhin erstellen wir ein ldfu-Programm, welches das Agentenverhalten definiert. Dieses liest die bereitgestellten Informationen der Artefakte aus und interagiert mit den Artefakten, welche wiederum auf die Anweisungen des Agenten reagieren.

Den kompletten Code zu diesem Beispiel finden Sie in unserem github Repository: Link

Bei der Umsetzung kann Ihnen, insbesondere als schnellen Visualisierung und zum Prototyping, der RDF-Browser als Plugin für den Firefox-Browser behilflich sein: Link

Umsetzung des Szenarios

Die einzelnen Schritte sind der Reihe nach angegeben. In Klammern ist die jeweilige Datei im Repository als Pfad angegeben (*in/dieser/Art*), in welcher der referenzierte Code komplett einsehbar ist.
Beachten Sie auch die Best Practices direkt nach diesem Kapitel, welche Anmerkungen zum Umgang mit den Systemen und bei Fehlverhalten bereitstellen.

1. BOLD aufsetzen

BOLD simuliert das Verhalten der Artefakte. Die .trig-Datei gibt dabei die Ausgangslage und verfügbaren Endpunkte vor.

1. Initiale Dateien als .trig-Datei (*bold/data/shopfloor.trig*)
    a. Definiere ShopFloor mit Größe 7x7, 3 Transporter, 4 Workstations und Farben (green, blue, yellow, red)
        i. Erstelle Named Graphs für Artefakte, bspw /station/green/
        ii. Fülle für die Artefakte das RDF auf, bspw. arena:locationX und arena:locationY zum Definieren der Position

iii. Erstelle NamedGraphs für die Unterressourcen der Artefakte bspw. /station/green/events oder /transporters/4/tasks
iv. Erstelle NamedGraph /colors/ unter welchem die Farben der Stationen und Produkte liegen
v. Erstelle NamedGraph /productsDelivered/ unter welchem die Anzahl der gelieferten Produkte nach Farben angegeben ist

b. Definiere LDP-Container welche die Transporter, Workstations und Farbmarker enthalten, damit diese auch später gecrawlt werden können
c. füge Transporter-Container, Workstation-Container und ShopFloor zu </> (=Root) hinzu

Die sim.ttl-Datei definiert die Parameter der Simulation und wird zum Starten bewusster genutzt.

2. Erstelle sim.ttl zum Starten von BOLD (*bold/data/sim.ttl*)
    a. Definiere initialen Zeitpunkt, die Dauer der Zeitschritte, Anzahl der Zyklen und ggf. Seed für Zufallszahlerzeugung
    b. Prüfe Korrektheit der Initialen Daten mit PUT von sim.ttl auf BOLD (Endpunkt http://localhost:8080/sim) um die Simulation zu starten

init.rq erstellt die Startpositionen der Transporter und die Felder des Shopfloors

3. Erstelle initiale SPARQL Requests (*bold/queries/init.rq*)
    a. Randomisieren von Transporterpositionen innerhalb der gültigen Grenzen
    b. Erstellen aller gültigen Koordinaten-Kombinationen und Binden als URIs für die Shopfloor Tiles
        i. Die Graphen Farbmarker pro Tile werden hier ebenfalls gleich angelegt und vorinitialisiert mit dem Wert 1000 (ein Wert, welcher ein Feld als “noch nicht markiert” kennzeichnet)
        ii. Die Kombination der möglichen Koordinaten-Pairing finden über VALUES statt
        iii. im unteren Bereich der SPARQL-Query erstellen wir die zugehörigen URIs aus den erzeugten String

In update-stations.rq wird während der Simulation das Verhalten der Station in SPARQL definiert, darunter wie Produkte erzeugt und vernichtet werden.

4. Workstationverhalten (aka Black Box) (*bold/queries/update-stations.rq*)
    a. Pro Station (grün/gelb/rot/blau) muss der Produktionsprozess in SPARQL definiert werden, damit die Farbzuweisung stimmt, d.h. dass rote Stationen nur blaue/grüne/gelbe Produkte produzieren oder grüne Stationen nur blaue/rote/gelbe Produkte
    b. Wenn der Outputport leer ist, erstelle eine UUID und verlinke mit arena:products vom Outputport zum Produkt
        i. die UUID wird zufällig mit STRUUID erzeugt
        ii. per FILTER kann ein fixierter Wert für eine Zufallszahl angegeben werden, damit die Station nur zufällig Produkte erzeugen und auch manchmal leere Transporter rumfahren können
    c. Das erstellte Produkt erhält eine Farbe, welche nicht die Farbe der erstellenden Station ist

        i. Die geschieht indem man ein Sample aus den definierten Farben ziehen lässt, wobei die gezogene Farbe nicht jener der Station entsprechen darf
    d. Das weitere Befüllen von Stationen passiert über das Verhalten von oben automatisch (wenn ein Transporter ein Produkt aufgenommen hat, wird somit direkt ein neues erzeugt)
    e. Prüfe den Inputport der Station, ob ein Produkt darin
        i. wenn die Farbe des Produkts jener der Stations entspricht, lösche das Produkt und erhöhe den Counter für die Anzahl gelieferter Produkte um 1
        ii. wenn nicht, tue nichts. Der Inputport bleibt befüllt.

In update-transporter.rq wird während der Simulation das Verhalten der Transporter in SPARQL definiert, darunter wie Bewegungstasks ausgeführt werden und Produkte aufgenommen / abgelegt

5. Reaktives Grundverhalten für Transporter (*bold/queries/update-transporters.rq*)
    a. Transporter haben einen Container, in welchem arena:TransporterTask gespeichert werden
    b. Transporter sollen einen arena:TransporterTask vom Agenten erhalten, welcher angibt, auf welches benachbarte Feld sich der Transporter bewegen soll
        i. Während der Transporter Bewegungen ausführt, bekommt dieser einen Status arena:busy. Somit wird verhindert, dass der Agent während der Bewegung zusätzliche Tasks schickt (der Agent schickt nur Tasks an Transporter mit Status arena:idle).
        ii. Wenn der Transporter steht und keine Tasks mehr hat, ist der Status arena:idle. Er kann somit neue Tasks erhalten.
        iii. Transporter stellen sicher, dass der Agent nur valide Felder innerhalb der definierten Shop Floor Größe zur Auswahl bekommt und prüfen ihr aktuelles Feld und ihre umgebenden Felder.
        iv. Wenn der Transporter benachbarte Felder wahrnimmt, welche außerhalb der Grenzen des Shopfloors liegen würden, so gibt er diese Felder nicht mit Koordinaten an, sondern als arena:nil
            1. Bsp: Ein Transporter steht auf (0,0). Sein östliches Feld ist somit per URI gegeben als shopfloor/1/0/ , sein nördliches (theoretisch -1, 0) als arena:nil
    c. Wenn ein Agent einen Task zur Bewegung schickt (TransporterTask), wird dieser entsprechend ausgeführt und der Transporter bewegt sich an diese Stelle
        i. Transporter dürfen sich nur auf ein Feld bewegen, wenn dort kein anderer Transporter steht, und das Feld innerhalb der Grenzen. Ist das nicht der Fall, wird der Task gelöscht.
        ii. Transporter leeren nach der erfolgreichen Ausführung eines Tasks den gesamten Task Container
    d. Der Agent besitzt einen “Sanity-Check” mit dem geprüft wird, ob ein Bewegungs-Task ausführbar ist. Wenn nicht (bspw Feld existiert nicht oder Wert ist arena:nil), wird der Bewegungs-Task gelöscht.

e. Transporter berechnen ihre umgebenden Floor Tiles von der aktuellen Position aus für alle Himmelsrichtungen (N, NE, E, SE, S, SW, W, NW)
    i. da der Shopfloor mit einfachen 2D-Koordinaten ausgelegt ist, wird in jeder zugehörigen Regel nur die aktuelle x-, y-Koordinate erhöht / verringert und hierdurch die URI der zugehörigen Nachbarfelder gebunden
f. Wir definieren zu debugging Zwecken transporterinterne Counter
    i. arena:discardedTasksInvalid - beschreibt die Anzahl der verworfenen Tasks (bspw. weil das angefragte Ziel-Tile kein benachbartes Tile des Transporters war)
    ii. arena:discardedTasksBlocked - beschreibt die Anzahl der Tasks welche der Transporter hätte ausführen können, wenn das Ziel-Tile nicht durch einen anderen Transporter blockiert gewesen wäre
    iii. arena:succesfulMov - beschreibt die Anzahl der erfolgreichen TransporterTasks welche ein Transporter ausführen konnte
g. Wenn sich Transporter auf demselben Tile wie eine Station befindet
    i. Prüfe ob Output-Port der Station ein Produkt enthält. Wenn ja, nehme Produkt “auf”” (setzte Objekt des Transporters arena:product auf product URI) und lösche URI bei Station / setzte sie auf arena:nil
    ii. Prüfe ob Station die gleiche Farbe hat wie das Produkt, welches der Transporter ggf referenziert. Wenn ja, “lege” Produkt in Input-Port der Station (setze Referenz zu Produkt für Input-Port) und lösche URI bei Transporter / setzte sie auf arena:nil

2. ldfu Agentenverhalten definieren

Der ldfu Agent steuert in followMarkers.n3 die Bewegung des Transporter-Artefakts. Er wertet die für den Transporter sichtbaren Felder aus, erkennt, ob Stationen in der Nähe sind und veranlasst das Setzen / Verändern und Folgen für Stigmergie-Marker aller definierten Farben.

1. Agent für Transporter (*ldfu/followMarkers.n3*)
    a. Alle Komponenten des Shopfloors müssen von ldfu per GET abgerufen werden
        i. Die root muss angegeben werden, bspw. <http://127.0.1.1:8080/>
        ii. Produkte, Stationen, Tiles, Marker müssen abgefragt werden, bspw über ldp:contains-Beziehungen oder andere Prädikate
    b. Transporter folgen in der gesuchten Farbe ihre Produkts mit absteigendem Wert
        i. nehme alle 8 Nachbarfelder und verlange eine >= Relation zwischen den Werten der Marker (alle entweder vorinitialisiert mit 1000, wenn im Feld, 1001 wenn außerhalb, oder kleiner wenn zum Ziel führend)
        ii. Bei gleichen Werten der Marker soll der Transporter ein beliebiges der Nachbarfelder nehmen (Exploration)
    c. Zwei Bewegungsarten des Transporters (beachte die im .trig-File definierten Grenzen des Shopfloors bei der Bewegung):
        i. Random Walk (= keine Stigmergiemarker in der Nähe des Transporters): Zufällig zu einem Feld neben dem Feld Transporter (N,

NE, E, SE, S, SW, W, NE). Eine Regel wird als erstes matchen und daher einen entsprechenden POST-Request an den Transporter mit einem TransporterTask schicken.

ii. Stigmergie (= folge dem Gradienten der Stigmergiemarker): Gezielt zu einem benachbarten Feld (N, NE, E, SE, S, SW, W, NE) schicken, von dem sich der Agent eine Verringerung der Distanz zur nächsten Station verspricht (da der Wert des Gradienten kleiner ist)
    1. Nimm alle Tiles auf deren Marker die arena:color Farbe des Produkts enthält
    2. Suche nach dem Tile, welches den kleinsten (oder einen von mehreren kleinsten) Wert hat und setzte dieses als Ziel (sortiert nach der Art tile1 <= tile2 <= tiles3 … → tile 1 wird Ziel).
        a. Dies muss für alle möglichen Matchings in ldfu ausgeschrieben werden, heißt eine Regel falls tile1, die kleinste wäre, eine für tile2, eine für tile3…
    3. Schicke per POST-Request als TransporterTask an Transporter

d. Tiles werden mit Stigmergie-Markern versehen
    i. Auf den Feldern liegen Marker der Farbe mit einem zugeordneten Wert
    ii. Alle Felder werden für alle Farben mit dem Wert 1000 initialisiert (in BOLD *bold/queries/init.rq*)
    iii. Unzugängliche Felder bekommen einen ldfu-internen, temporären Stigmergie-Wert zugewiesen (dieser wird <u>nicht</u> nach BOLD geschrieben oder gespeichert)
        1. Tiles auf denen ein Transporter steht bekommen den temporären Stigmergie-Wert “1001” für alle Farbmarker
        2. Tiles die außerhalb des Shopfloore sind bekommen den temporären Stigmergie-Wert “1002” als DummyMarker (da das Feld nicht existiert, gibt es auch keine Ressource dazu, welche der Marker ist - wir erzeugen diese temporär intern als dummy)
        3. beide Werte signalisieren ausgesprochen ungünstige Stigmergie, welche der Agent ignoriert. Stattdessen werden immer lieber Felder mit Stigmergie “1000” vorgezogen (d.h. mit initialem Wert) oder die mit bereits gesetzten Wert unter 1000.
    iv. Wenn der Agent einen Transporter auf demselben Feld wie eine Station erkennt, erhält dieses Feld den Stigmergiewert 0:
        1. ein PUT-Request mit 0 wird immer auf die Marker-Ressource des Felds mit der gleichen Farbe wie jener der Station geschrieben, um das globale Minimum des Farbgradienten zu signalisieren
    v. Agent prüft umgebenden Felder der Transporter nach dem Wert aller ihrer Stigmergie-Farben. Wenn ein Marker eines benachbarten Felds einen geringeren Wert besitzt, als der Marker des derzeitigen Felds, bekommt das Feld des Transporters diesen Wert+1 für die entsprechende Farbe. Somit baut sich ein absteigender Gradient auf, dessen globales Minimum 0 an der Position einer Station ist, und dem

der Agent nur schrittweise absteigend folgen muss, um zu dieser Position zu kommen

1. nimm Marker einer Farbes des Feld des Transporters, und ein beliebiges benachbartes Feld (N, NE, E, SE, S, SW, W, NE), dessen Wert ?value für diese Farbe kleiner ist
2. errechne mit (?value "1"^^xsd:integer) math:sum ?valuePlusOne einen neuen Wert und vergleiche, dass dieser auch kleiner ist als der Wert, welcher vorher für diese Farbe als Wert hinterlegt war
3. Sende einen PUT-Request an die Stigmergie-Marke und überschreibe den alten Wert der Farbe mit dem neuen Wert, der nun kleiner ist als der vorherige, aber dennoch größer als der Wert des Nachbarn
4. → hieraus bildet sich der Gradient

Best practices

Was tun wenn nichts funktioniert? Und worauf ist allgemein zu achten?
Verschiedene Ansätze:

- RDF Dumps nach Ende der Simulation anschauen in BOLD im bold.replay.dump Verzeichnis
- URIs prüfen hinsichtlich Schreibfehlern. 90% aller Fehler sind idR von Schreibfehlern kommend (bspw. ein “/” vergessen; Plural und Singular verwechselt)
- mehrere Queries in einem SPARQL File müssen durch Semikolon getrennt werden
- die Standard SPARQL Zufallsfunktion RAND() ist verbuggt, da RDF4J Funktionen ohne Arugment nur einmalig auswertet. Daher benutzen wir die BOLD eigene Funktion sim:rand() mit einem beliebigen Argument, um Zufallszahlen in [0,1) zu erzeugen
- beim Verknüpfen von Strings mittels CONCAT sollte darauf geachtet werden, dass alle Argumente auf String gecastet werden oder per se Strings sind
- BOLD setzt den Einstiegspunkt der URIs auf **127.0.1.1** - also **NICHT** 127.0.0.1
- Falls Sie bedingte Ausführungen in SPARQL benutzen (bspw. zum Abfragen ob der Transporter idle oder busy ist, bedenken Sie, dass Sie einerseits in der Where-Clause die Condition abfragen und andererseits in der Insert und Delete Clause den neuen Zustand setzen bzw. den alten löschen.
- Beachten Sie die spezielle Semantik von SPARQL Keywords, wie bspw OPTIONAL (dieses ist etwa nur erfüllt, wenn alle angegebenen Tripel gefunden werden)
- der ldfu Parser kann keine Literale wie Ganzzahlen oder Boole’sche Werte, diese müssen bspw. als “0”^^xsd:integer oder “true”^^xsd:boolean angegeben werden. Hiezu muss auch der entsprechede Prefix für XMLSchema geladen werden.
- BOLD weicht bei der Verwendung von UNION als SPARQL Funktion vom Standard ab und fordert, dass entsprechende gleiche Requests, welche gemacht werden, in beiden Subqueries gemacht werden müssen (d.h. sie werden einfach in beide Teile kopiert)
- Die Ausgabe von ldfu als ntriples kann auch direkt in fuseki geladen werden, um bestimmte Triple-Patterns per SPARQL zu finden und zu visualisieren

- Alle URIs welche von ldfu ausgewertet werden sollen, sollten auch im Agenten-Programm per GET angefordert werden, um sie bspw. später im Programm verwenden zu können (das beschreiben wir bspw in 2.1.a des ldfu Programms oben)
- log:dtlit kann kann ein Casting von Stirng nach Integer vollführen; math arbeitet mit und gibt am Ende hingegen immer Strings aus
- nach dem Beenden von BOLD mittels Interrupt kann BOLD nicht sofort neu gestartet werden, da der Port noch belegt ist. Nach einer Wartezeit von 30sec-1 min sollte dies wieder funktionieren.
- blank nodes können zwar auch benutzt werden, um Tripel abzuspeichern, jedoch ist Verarbeitung mit ldfu deutlich einfacher, wenn Ressourcen benutzt werden