Variability Extraction and Analysis Toolkit (VEXA)

VEXA Introduction

The Variability Extraction and Analysis (VEXA) toolkit is a collection of complementary procedures to help with many different tasks of **variability extraction**, **feature analysis**, **visualization** and the calculation of **user-defined metrics**. Implemented as a plug-in for the "world's leading Graph Database" Neo4j , the VEXA toolkit leverages the powerful graph storage and processing capabilities of Neo4j to enable **detailed dependency analyses** of source code artifacts (e.g., #ifdef variability in C/C++ code) and **reveal intricate feature connections** across project artefacts along with graph visualization possibilities.

The Cypher Query Language

Cypher is a declarative query language - simple, nevertheless very powerful - that allows for expressive querying and efficient manipulating of a property graph. The VEXA toolkit extends Cypher using the concept of user-defined procedures and functions to provide extraction and analysis procedures.

All VEXA Cypher queries can be displayed by running the query:

```
CALL vexa.help("vexa")
```

The Property Graph Model of VEXA

During the extraction and analyses steps, VEXA generates results and stores them directly in the Neo4j graph data base. The storage model of Neo4j is the *Property Graph Model*, which is represented by a set of *nodes* and *relationships* that can both hold any number of attributes (key-value-pairs) called properties.

Setup Database

The first step in running a variability analysis with VEXA, is to setup the Neo4j graph database. You can create constraints on node properties and setup indices to speedup search queries.

Create Constraints

We are going to create a simple constraint which states that every project name should be unique.

```
//== Constraints
CREATE CONSTRAINT ON (p:Project) ASSERT p.name IS UNIQUE;
```

Create Indices

Now it is time to create indices on node properties. For our use case we create an index for the `name, uri, condtion, type` properties of the respective graph nodes.

```
//== Indices
CREATE INDEX ON :File(name);
CREATE INDEX ON :File(uri);
CREATE INDEX ON :File(extension);
CREATE INDEX ON :Dir(name);
CREATE INDEX ON :Dir(uri);
CREATE INDEX ON :VariationPoint(condition);
CREATE INDEX ON :VariationPoint(type);
CREATE INDEX ON :Symbol(name);
```

Clear Database

Before we run any extraction or analyses steps, were are going to clear the database by removing all graph nodes and graph edges.

```
//== Clean graph database
MATCH (n) detach DELETE (n);
```

ERIKA3 Use Case

VEXA is use case driven. An use case is represented by a project node (`:Project`) with a unique `name` and `uuid` property.

We demonstrate the functionality of VEXA by performing various extraction and analysis steps on the ERIKA3 source code.

ERIKA Enterprise is a Real-Time Operating System (RTOS) made by Evidence Srl. It implements the AUTOSAR OS and OSEK/VDX API specification. ERIKA Enterprise is used in production in various automotive and white goods applications.

Create Project

The first step is to create a project and give it a name. We use a form field here:

ERIKA3-UseCase1

```
//== Create VEXA infrasturcture for projects
CALL vexa.project.create('ERIKA3-UseCase1') YIELD node
RETURN node.name AS `Project`
```

By calling the VEXA procedure `vexa.project.create()` we generated the `:Project` node, which represents our use case.

Add Resources

The next step is to add resources (:Resource) to our project. A resource is a software engineering asset that can be located by an universal identifier (URI). In the current version, VEXA only supports resources located on the filesystem.

file:/C:/Users/paule/Coding/use_cas

We add the path of the ERIKA3 source code repository by running the query:

```
CALL vexa.project.add.resource('ERIKA3-UseCase1', 'file:/C:/Users/paule/Coding/use_cases/ERIKA3/erika3');
```

The `vexa.project.add.resource()` procedure will geneate a new `(:Resource)` node and link it with the `(:Project)` via the relationship `[:hasLink]` . We can inspect the generated link by running the query:

```
MATCH p=()-[r:hasLink]->() RETURN p LIMIT 1
```

Initialize Project

In the initialization phase, VEXA will analyze all `(:Resource)` nodes linked to the project. Depending on the type of the resource, VEXA will try to adapt `(:Resource)` nodes by attaching `(:Adapter)` nodes.

```
//== Initialize project
CALL vexa.project.init('ERIKA3-UseCase1');
```

There are various VEXA adapters specialized for a specific purposes. In our case VEXA will adapt the root directory of the ERIKA3 source code by running a `FileSystemAdapter` that is responsible for analyzing the file tree structure of a directory. Every VEXA adapter takes as input a resource node and generates an abstraction of the resource by decomposing it into atomic elements and assembling them into a graph representation.

We can display the generated abstraction by running the query:

```
MATCH p=()-[:ADAPTED_BY]->()-[r:GENERATES]->() RETURN p LIMIT 25
```

All nodes generated by a VEXA adapter are linked to the `(:Adapter)` node via the relationship `[:hasOrigin]`.

Extraction

We take a quick tour to checkout the extraction capabilities of VEXA. For variability analyses of the ERIKA3 source code, we are interested in the **compile time variabilty** introduced by `#ifdef` preprocessor statements. VEXA includes an adapter for extracting `#ifdef` variability information in C/C++ source code.

Run Variability Extraction

Let's recall the methodology behind VEXA. VEXA uses adapters to decompose software engineering assests, represented by `(:Resource)` nodes, and assemble them into an abstract graph representation for later processing. To extract `#ifdef` variability from all source files we first need to find them.

A simple Cypher query will match all resources with a file extension matching the regular expression `'^(c|h|cc|hh)$'`. To see the results we can run the query:

```
//== Find all source files
MATCH (f:File) WHERE f.extension =~ '^(c|h|cc|hh)$'
RETURN f.name, f.extension, f.uri
```

Now it is time to run the extraction process by adapting all source files. VEXA will select the correct adapter based on the file extension. For C files VEXA will instantiate the `SrcmlCAdapter`, which will generate an abstract representation of the source code encompassing all variabiltiy information. We can trigger the extraction phase for all C source files by running the query:

```
//== Run SrcmlAdapter on all C, C Header files in batches
MATCH (f:File) WHERE f.extension =~ '^(c|h|cc|hh)$'
WITH collect(f) as files
CALL vexa.resource.adapt.iterate(files, {batchSize:4, parallel:true})
RETURN size(files) AS `Processed source files`;
```

The procedure `vexa.resource.adapt.iterate()` will process all matched source files and generate the abstract representations for each file. Source files are processed in batches to speed the overall extraction process.

While the extraction process is running we can explore data model we employ for variability analyses.

Code Metrics

A great deal of analyses can be written in pure Cypher. VEXA contains a collection of useful procedures to do the

work for you.

Generating code metrics is one way to get first insights into the code base and assess different aspects of code variability.

VEXA implements the procedure `vexa.metrics.cpp()` to generate metrics for *C preprocessor (CPP)* usage based on the work of:

> *Hunsen, Claus, et al. "Preprocessor-based variability in open-source and industrial software systems: An empirical study." Empirical Software Engineering 21.2 (2016): 449-482.*

We can generate the CPP metrics by running the query:

```
CALL vexa.metrics.cpp({})
```

A lot of VEXA procedures can be fine-tuned to refine an analysis. To refine the generation of CPP metrics we want to **exclude** C/C++ header guards and only consider variation points in the source code that emerge from `#ifdef, #ifndef` preprocessor statements. To achieve this analysis goal, we set the `headerGuardFilter` and `vpTypes` options for the `vexa.metrics.cpp()` procedure.

```
CALL vexa.metrics.cpp({headerGuardFilter:'.*_(H|H_|h|h_)$', vpTypes:['IFDEF', 'IFNDEF']})
```

Custom Analyses

Because VEXA is a part of the Neo4j graph database, the user can leverage the full spectrum of graph processing algortihms to write custom analyses. The VEXA adapters did the heavy lifting for analyzing the filesystem and the source files.

Now it is up to the user to utilize the genearted abstractions in the graph database. Exploring the graph database and writing custom analyses is an **iterative** and **iteractive** process.

The Neo4j browser is a good starting point to try things out but the visualization capabilities are quite limited. Thus, other visualization front-ends can be used to assist the engineer, programmer or analysist with that task. **FeDeV** by ScopeSET is a tool that offers more sophisticated visualization support.

All custom analyses can be written using Cypher queries and VEXA procedures/functions.

Analysis Goal

For our first custom analysis we are interested in the nesting depth of variation points. We want to know which configuration constant (preprocessor symbol) has the highest nesting level.

Nesting Depth Analysis

We start our analysis with an overview of preprocessor symbols and their usage in the ERIKA3 source base.

```
MATCH (s:Symbol)<-[:HAS_SYMBOL]-(vp:VariationPoint)
WITH DISTINCT s.name as symbolName, collect(DISTINCT vp) as vps
RETURN symbolName as `Preprocessor Symbol`, size(vps) as `Usage Count`
ORDER BY size(vps) DESC
```

The query returns a table with all preprocessor symbols and the number of variation points they are used in.

Table 1. Analysis iteration 1

Preprocessor Symbol	Usage Count
…	…

As we can see the symbol `OSEE_TC_COMPL_INTTAB` is used in more than 3000 variation points.

Now we are interested in how this preprocessor symbols are distributed among source files. We extend our previous query to collect all the file names the preprocessor symbol is used in.

```
MATCH (s:Symbol)<-[:HAS_SYMBOL]-(vp:VariationPoint)-[:CONFIGURES]->(cb:CodeBlock)-[:hasOrigin]->(:Adapter)<-[:ADAPTED_BY]-
(f:File)
WITH DISTINCT s.name as symbolName, collect(DISTINCT vp) as vps, collect(DISTINCT f.name) as files
RETURN symbolName as `Preprocessor Symbol`, size(vps) as `Usage Count`, files as `Source Files`
ORDER BY size(vps) DESC
```

The generated table has an additional column where the names of the source files are displayed.

Table 2. Analysis iteration 2

Preprocessor Symbol	Usage Count	Source Files
…	…	…

In the last step of our nesting depth analysis we want to know the maximum nesting depth of a code block that is enclosed by `#ifdef` statements, which use a specific configuration constants.

```
MATCH (s:Symbol)<-[:HAS_SYMBOL]-(vp:VariationPoint)-[:CONFIGURES]->(cb:CodeBlock)-[:hasOrigin]->(:Adapter)<-[:ADAPTED_BY]-
(f:File)
WITH DISTINCT s.name as symbolName, collect(DISTINCT vp) as vps, collect(DISTINCT f.name) as files,
collect(apoc.node.degree.in(cb, 'CONFIGURES')) as NDs
RETURN symbolName as `Preprocessor Symbol`, size(vps) as `Usage Count`, files as `Source Files`, apoc.coll.max(NDs) as `ND_max`
ORDER BY size(vps) DESC
```

Table 3. Analysis iteration 3

Preprocessor Symbol	Usage Count	Source Files	ND_max
…	…	…	…

As you can see, no VEXA specific procedures/functions were used to write the nesting depth analysis. Even though a variety analyses can be written in pure Cypher it requires tremendous experience and Cypher skills to fullfill complex analysis goals. The VEXA toolkit offers a variety of helper functions that support the user in writing custom analyses.

VEXA helper functions

- `vexa.metrics.sd`: calculate scattering metrics
- `vexa.metrics.td`: calculate tangeling metrics
- `vexa.metrics.nd`: calculate nesting depth
- …