Towards Applying Model-Transformations and -Queries for SOA-Migration

Tassilo Horn, Andreas Fuhr, Andreas Winter
{horn,afuhr,winter}@uni-koblenz.de

Abstract: In today's businesses, a lot of monolithic legacy systems exist. In order to allow for a higher flexibility and a better alignment to the ever-changing business processes, migration to service-oriented architectures is often intended. In most cases, this is done by wrapping the legacy system with an additional layer providing functionality as services. But this approach does not result in more flexibility, because the old system is unchanged. This paper proposes the extensive use of model-driven techniques, like model-querying and transformations, for migrating legacy assets into service-oriented architectures. The techniques used were evaluated in a first attempt at migrating parts of the functionality of the open-source tool GanttProject into a service.

1 Motivation

In today's IT world, the design paradigm of *service-oriented architectures* (SOA) gains utmost importance. Within this paradigm, systems and system topographies are built of interoperable services, which allow easy composition to systems supporting business tasks. When designing SOAs, one strives for a loose coupling between services, which is realized by slim interfaces used for communication between services that support only a certain number of messages defined in *service contracts*. With this approach, each service implementation could be replaced with another one, which satisfies the same service contract [OAS06].

According to SOA-enthusiasts, reusability, loose coupling, and the alignment of services to business processes offer much more flexibility to adapt to changes than traditional system landscapes, because adapting to business changes is understood as something natural which happens every now and then.

When introducing a SOA, one of the most important tasks is to integrate existing legacy systems. Currently, in industrial approaches this is done by exposing services which wrap existing functionality (e. g. [SL06]). But wrapping has several drawbacks:

- The integrated legacy system as well as the adapter layer have to be maintained,
- and changes in business processes may lead to adaptions in both the legacy systems as well as the wrapping code.

When wrapping existing systems, the SOA-benefits are only noticeable for service users, but for service providers, the complexity increases even more.

Another solution is to supersede each legacy system by a set of services created from scratch, which can be orchestrated in order to support the same business process or functionality. Of course, designing and implementing each service anew, with having SOA in mind, is often not feasible. Re-implementing software components, including functional adaptations is error-prone, time-consuming and success is not guaranteed [Sne97].

To leverage all of SOA's benefits, methods for designing, developing and introducing SOAs, like IBM's *SOMA* (*Service-Oriented Modeling and Architecture*, [AGA+08]) or Michael Bell's *SOMF* (*Service-Oriented Modeling Framework*, [Bel08]) were developed. These methods mainly focus on the initial development of SOAs, and do not reflect migration of already existing assets to SOA. At least the *SOMA* methods provides extension points for integrating other techniques and methods. How those can be exploited for reeingineering and migration is discussed in [FWGH09].

A coherent, model-driven approach to software migration, providing methods and techniques supporting (1) finding possible service candidates in existing systems, (2) transforming them into services and (3) orchestrating them in an at least semi-automatic fashion is missing.

In the just started soamig project[1], new methods and techniques for migrating legacy assets to SOA are to be explored. Model-driven techniques and techniques from software reengineering funding on graph querying and graph transformations are used for program comprehension, for finding service candidates in legacy code, and for migrating self-contained parts of legacy systems as services.

To get an consolidated view on all artifacts created or analyzed during a SOA migration (business process models, architectural models, legacy code, ...), and to enable integrated analysis, a well-defined metamodel encompassing all of those artifacts has to be created. One important aspect is that the metamodel as well as its instances contain traceability information ([SERW08], [Sch09]) between different artifacts. This enables queries like retrieving all components involved in handling a given business process, and detecting componts in code which implement business processes.

This paper is a first feasibility study and focusses on the code level to show some application areas for model-driven techniques in SOA migration. Section 2 explains how models are represented as TGraphs. Section 3 is focussing on querying models using the declarative query language GReQL, and in Section 4 the model-transformation framework GReTL is introduced. The application of these techniques in SOA migration is shown in the comprehensive example in Section 5. The scalability, performance and expressiveness of the described techniques is depicted in Section 6. Section 7 is devoted to related migration approaches. Finally, a short conclusion and outlook is given in Section 8.

2 Representation of Models as TGraphs

The representation of models is based on a general class of directed graphs called *TGraphs* [ERW08]. With TGraphs, the graphs themselves, all nodes, and all edges are typed and

[1] soamig project members are pro et con GmbH, Chemnitz, Amadeus Germany, Bad Homburg and the Institute for Software-Technology at the University Koblenz-Landau. The work is funded by the German Minister of Education and Research (BMBF) grant "SME innovative" 01IS09017C. `http://www.soamig.de`

attributed. Nodes and edges are orderd globally, and for each node all incident edges are ordered, too. Edges are viewed as first class citizen, and navigability is always bidirectional and does not depend on the edge's direction. The graph library *JGraLab* (*Java Graph Laboratory*) provides a convenient and efficient API for accessing and manipulating TGraphs.

The structure of TGraphs is specified by TGraph schemas, i. e. metamodels for classes of TGraphs. Such schemas are specified by using an UML-profile of class diagrams called *grUML* (*Graph UML*), a tool-ready subset of CMOF comparable to EMOF. In grUML diagrams, node and edge types and their attributes are specified by UML classes and associations. Multiple inheritance between both node and edge classes is supported.

An example grUML diagram of a small excerpt of the Java schema, specifying the different Java types, is shown in Figure 1. The arrows indicate the reading direction here.

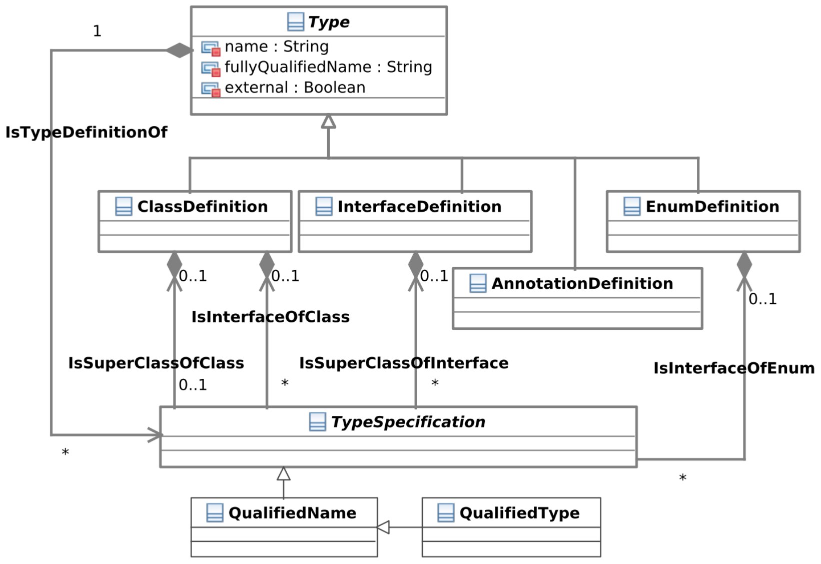


Figure 1: The part of the Java schema modeling types

The complete Java schema contains about 90 node and 160 edge types and covers the syntax of Java version 6. An appropriate parser creates TGraphs conforming this schema from Java source code, class and jar files. Additionally, a Java code generator is able to serialize graphs back to source code.

```
class HumanResourceManager implements ResourceManager, CustomPropertyManager {/* ... */}
interface ResourceManager {/* ... */}
interface CustomPropertyManager {/* ... */}
```

Listing 1: Sample Java source code

A simplified TGraph representing the sample code given in Listing 1 is shown in Figure 2. The graph depicts the interrelationships between a class HumanResourceManager and the two interfaces it implements.

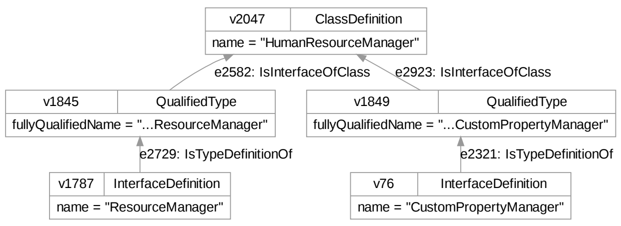

v2047
ClassDefinition
name = "HumanResourceManager"
e2582: IsInterfaceOfClass
e2923: IsInterfaceOfClass
v1845
QualifiedType
fullyQualifiedName = "...ResourceManager"
v1849
QualifiedType
fullyQualifiedName = "...CustomPropertyManager"
e2729: IsTypeDefinitionOf
e2321: IsTypeDefinitionOf
v1787
InterfaceDefinition
name = "ResourceManager"
v76
InterfaceDefinition
name = "CustomPropertyManager"

Figure 2: A simplified TGraph visualization of the Java code in Listing 1

Although this paper focusses on TGraphs representing Java source code, the general approach is applicable for any other language as well. In the soamig project, analysis and transformation of integrated TGraph models representing Java and COBOL code, system architecture, and business processes will be performed. Additionally, dynamic aspects and dependencies introduced by property files or deployment descriptors can be incorporated by enhancing the metamodel and providing appropriate parsers for the relevant artifacts, but this is out of the scope of this paper.

3 Querying Models with GReQL

One of the most important aspect in model-driven software migration is the ability to gather information which could not be gathered easily in the artifact's original form (i. e. source code). Graph-based methods and tools for program comprehension used in in the following were developed in the GUPRO project ([KW98, EKRW02]).

GReQL (*Graph Repository Query Language*, [BE08]) provides a declarative graph-query language, and its syntax bears some analogies to SQL. One of the most commonly used language elements is the *from-with-report* (*FWR*) clause. The `from` part is used to declare *variables* and to bind them to *domains*. In the `with` part, *constraints* can be imposed on the values of these variables. The `report` part is used to define the structure of the query result.

A sample query for retrieving all classes implementing the interface ResourceManager is depicted in Listing 2.

```
from e : E{IsInterfaceOfClass}
with startVertex(e).fullyQualifiedName =~ ".*\\.ResourceManager"
reportSet endVertex(e).name end
```

Listing 2: A GReQL query to find all implementors of ResourceManager

In the `from` part, the variable `e` is bound to all edges of type IsInterfaceOfClass. The constraint defined in the `with` clause requires that the fullyQualifiedName attribute of the edges start nodes ends with “ResourceManager”. In the `report` clause, the structure of the results is defined. The value of the name attribute of the class implementing that interface is reported. According to the graph in Figure 2, the “HumanResourceManager” is part of the result set, and it is its only element.

This query is also an example of efficient GReQL-queries. A trivial approach would be to define two variables iterating over nodes of type ClassDefinition and QualifiedType and to check, if an `IsInterfaceOfClass` relationship holds between them. That query’s effort would have been quadratic, whereas the query given here, is linear to the number of IsInterfaceOfClass edges.

One of GReQLs further features are *regular path expressions*, which can be used to formulate queries that utilize the interconnections between nodes. Therefore, symbols for edges are introduced: `-->` for directed edges, and `<->` if the direction is not considered. Additionally, an edge type written in angle brackets may follow the edge symbol. These symbols can be combined using regular operators: sequence, grouping (()), iteration (* and +), and alternative (|).

```
from outer, member : V{ClassDefinition}
with outer (<--{IsClassBlockOf} <--{IsMemberOf})+ member
reportSet member end
```

Listing 3: A query using regular path expressions to find member classes

Listing 3 shows a query for finding member classes. Two variables of type ClassDefinition are defined. The `with` clause tests, if the class definition bound to `member` is defined as member of `outer`’s class block. The + specifies the transitive closure, so all member classes defined in other member classes will be reported as well.

4 Transforming Models with GReTL

The *GReTL Graph Repository Transformation Language*, [EH09] is a Java framework for programming transformations on TGraphs. Instead of creating a new transformation language, including its own syntax from scratch, existing technologies were used, namely JGraLab’s Schema API for expressing imperative aspects and GReQL for expressing declarative aspects.

The GReTL transformations presented here use the most fine-granular means for specifying transformations. They can be seen as the bytecode and conceptual substruction of the abstract transformation machine. On top of that, generic higher order transformations can be created to ease the creation of complex transformations in a given context. A in-depth discussion of GReTL is given in [EH09].

In GReTL, a transformation is simply a subclass of the abstract Transformation class. Inside its transform() method, it calls operations derived from its superclass. These operations are aligned to methods of JGraLab’s schema API and thus to the JGraLab metaschema. They allow the creation of target schema elements, but additionally their signature contains parameters for specifying which instances of the created schema element should be created in the target graph by providing GReQL queries. Thus, the schema creation is done imperatively in plain Java, and the creation of schema element instances is done in a declarative fashion using GReQL.

To introduce GReTL, a very basic transformation is used. It creates a simple target schema, which contains only one node type uml.Class and an edge type uml.HasMemberClass. On the instance level, the target graph should contain one Class node for each ClassDefinition in the Java source graph. For each member class (a class defined inside another class), a HasMemberClass should be created, pointing from the owner class to the member class.

The operation invocation for creating the node class in the target schema and its instances is depicted in Listing 4. The first parameter specifies the fully qualified name of the new node class to be created in the target schema. So the name of the new node type is Class, and it is located in the package uml. The GReQL query given as second parameter is evaluated on the source graph and should return a set. Here, the set of all ClassDefinition nodes in the source graph is retrieved. The elements of this set are used as *archetypes* for the nodes to be created in the target graph, e. g. for each class definition node in the result set a new uml.Class node is created in the target graph.

```
VertexClass cls = createVertexClass("uml.Class", "V{ClassDefinition}");
```

Listing 4: GReTL operation invocation for creating the node type uml.Class and instances thereof

The mappings from source graph *archetypes* to their *images* in the target graph are saved and managed by the transformation framework. Additionally, those mappings can be used in following transformation operation invocations via the variables `arch_uml$Class` and `img_uml$Class`. The image mapping is the inverse of the archetype mapping. The transformation framework ensures bijectivity, so that they can be used to navigate from source to target graph and back again.

Listing 5 shows an example GReTL operation invocation, which creates a new edge type named HasMemberClass in the target schema's uml package.

```
EdgeClass knows = createEdgeClass("uml.HasMemberClass", cls, cls,
      "from outer, member :V{ClassDefinition} "                 // edge archetype set
    + "with outer <--{IsClassBlockOf} <--{IsMemberOf} member "
    + "reportSet outer, member end",
    "from t : $  reportMap t, nthElement(t, 0) end",            // edge start function
    "from t : $$ reportMap t, nthElement(t, 1) end");           // edge end function
```

Listing 5: GReTL operation invocation for creating edge classes and instances thereof

Both source and target node instances have to be instances of the uml.Class node type created in the former operation invocation. As archetypes set, the first GReQL query returns 2-tuples of ClassDefinition nodes. The constraint in the *with* part restricts the tuple elements to pairs, where the first component is the class definition containing the member class, and the member class itself is the second component. For each of those tuples, a new HasMemberClass edge will be created in the target graph. Like with `createVertexClass()` the mapping from archetypes to their images and the inverse function is exported to following operation invocations via the variables `img_uml$HasMemberClass` and `arch_uml$HasMemberClass`.

In order to create an edge, the source and target nodes have to be known. Their specification is done with the last two queries in Listing 5. The special variables `$` and `$$` are only syntactic sugar and refer to the result of the previous GReQL query and the result of the query before the previous one, respectively. So in both cases they access the result of the archetypes query's result.

The second query returns a function mapping the archetypes of this edge class to the archetypes of the nodes in the target graph, which should act as source for the new edges. The third query does the same for the target nodes. The archetype set for the HasMemberClass edge class contains tuples (o,m), where m is a member class of o. The function returned by the second query contains mappings $(o,m) \mapsto o$ for all (o,m) tuples in the archetype set, and the function returned by the third query contains mappings $(o,m) \mapsto m$. Both o and m are ClassDefinition nodes in the Java source graph. The transformation framework selects their images in the target graph as start and end node for the new edge to be created in the target graph, and those are uml.Class nodes created by the GReTL operation invocation in Listing 4.

The createEdgeClass() method is overloaded several times. The variant depicted in Listing 5 is the most compact one. Other variants allow for specifying role names as well as multiplicities. Additionally, there are GReTL operations for creating aggregation and composition edge types, and for creating attributes for node and edge types.

In the next section, a more complex transformation is shown. It recovers some architecture descriptions out of a graph representation of Java source code.

5 Model-Driven Migration

This section discusses the application of the previously introduced graph query and transformation techniques to discover and extract functionality from the open-source project scheduling and management tool GanttProject[2] in order to integrate it as a service in a service-oriented architecture.

While exploring the GUI of GanttProject, the functionality of managing resources was discovered. This feature may be of interest for the target SOA. Thus, this functionality will be made available as a service in the new environment. To do so, one needs to identify the classes and interfaces in the code implementing this feature. In order to be able to extract those parts as self-contained unit, their dependencies have to be resolved. After that, the actual transformation into the target architecture can be performed.

As a preparative step to enable model-driven migration, the legacy system is transformed into a TGraph model. In this example, only the Java source code is available. The transformation is done with the parser developed in [BV09]. It creates a TGraph representation of the source code, which conforms to a fine-grained Java 6 TGraph schema (Figure 1). The resulting graph contains 274.959 nodes and 552.634 edges.

[2]http://www.ganttproject.biz

Analyzing the legacy system. Identifying source code fragments to be migrated to the new SOA, requires to find proper code artifacts as starting points. Since "Resource Management" characterizes the service to be identified, a search for classes and interfaces named alike is appropriate. The GReQL query in Listing 6 finds all Types whose name contains the substring "*resource*manage*".

```
from t : V{Type} with t.name =~ ".*[Rr]esource.*[Mm]anage.*"
reportSet typeName(t), t.name end
```

Listing 6: GReQL: find all types matching "*resource*manage*"

The query result contains only the two tuples (InterfaceDefinition, ResourceManager) and (ClassDefinition, HumanResourceManager). The interface ResourceManager seems to provide what is needed for the target service. The query in Listing 7 retrieves all classes which implement this interface.

```
from e : E{IsInterfaceOfClass}
with startVertex(e).fullyQualifiedName =~ ".*\\.ResourceManager"
reportSet endVertex(e).fullyQualifiedName end
```

Listing 7: GReQL: find all implementors of ResourceManager

The result shows, that HumanResourceManager is the only implementor, so the focus is set on this class.

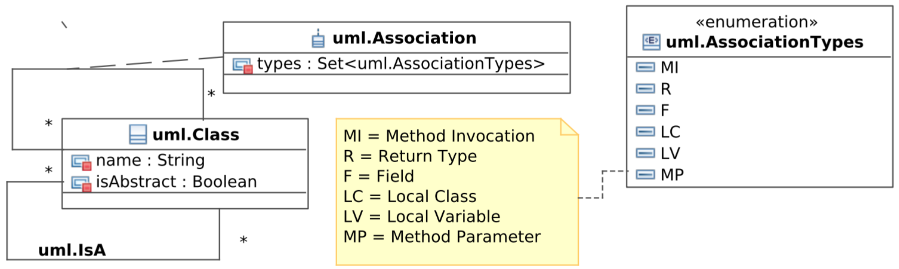


Figure 3: The target schema created by the tranformation

Transformation-based architecture recovery. The next task is to get an overview of the architectural component containing the HumanResourceManager. A simple class diagram, which shows it together with all other classes and used interfaces is appropriate here. This architecture recovery is done by a GReTL transformation. The target graph's schema is depicted in Figure 3. The architecture will only contain classes, generalizations (IsA) and associations. The types attribute associated to the association edges specifies the kind of usage. A part of the resulting UML graph is shown in Figure 4.

In the middle, a Class node (v2) representing the HumanResourceManager can be found. The IsA edge between it and the abstract Class depicting the ResourceManager (v12) shows the generalization hierarchy. The Association edges visualize usages, and their types attributes is set according the type of usage. For example, CustomPropertyImpl (v5) is a local

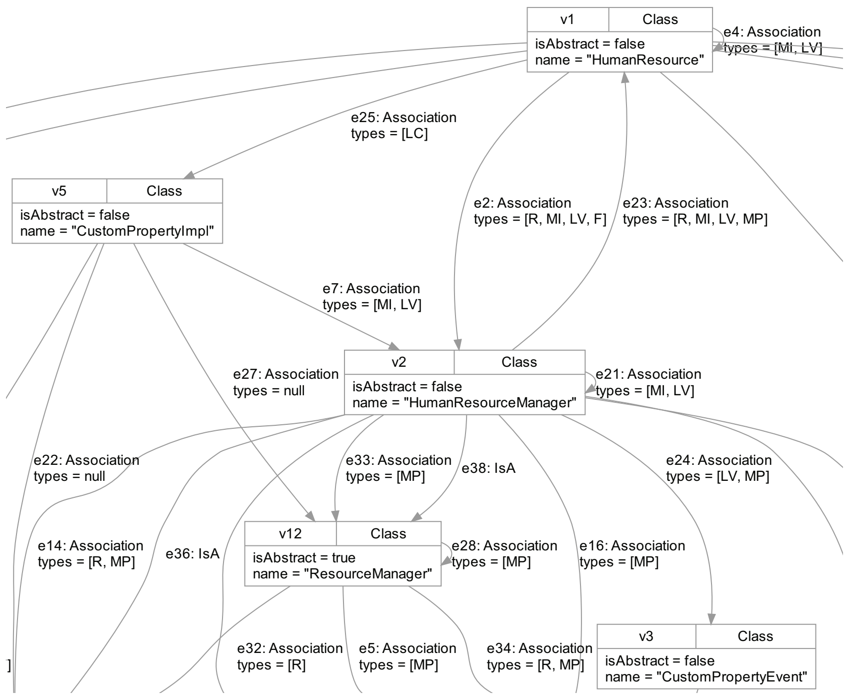


Figure 4: Partial result from architecture recovery

class (LC) defined inside HumanResource (v1), and HumanResource uses the HumanResourceManager (v2) by having methods returning an object of that type (R), by invoking methods of it (MI), and by having local variables (LV) and fields (F) of type HumanResourceManager.

The operation invocation for creating the node type Class and its instances is given below.

```
VertexClass umlClass = createVertexClass("uml.Class",
        "  from t : V{Type}                                              "
      + "with t.name =~ \".*HumanResourceManager\" or                    "
      + "  not(isEmpty(t [-->{IsTypeDefinitionOf}]                        "
      + "     -->{^IsBreakTargetOf,^IsContinueTargetOf,^IsTypeDefinitionOf}*  "
      + "     -->{IsMemberOf} (-->{IsClassBlockOf} | -->{IsInterfaceBlockOf})  "
      + "      & {hasType(thisVertex, \"Type\")                           "
      + "         and thisVertex.name =~ \".*HumanResourceManager\"}))    "
      + "reportSet t end");
```

A new vertex type uml.Class is created in the target schema. For each Type in the Java source graph, which either *is* the HumanResourceManager or which *uses* the HumanResourceManager via the complex regular path description, a new Class node is created in the target graph.

The name attribute for the newly created Class nodes is set by the following operation invocation.

```
createAttribute("name", umlClass, createStringDomain(),
      "from t : keySet(img_uml$Class) reportMap t, t.name end");
```

An attribute name is added to the target node class referenced by umlClass. The value domain is set to string. The last parameter is a GReQL query resulting in a map from Class archetypes (Type instances in the source graph) to the desired class name. The new Class nodes are named according to the source types they were created for.

The last operation invocation introduced here creates the IsA edge type and its instances.

```
createEdgeClass("uml.IsA", umlClass, umlClass,
  "  from super : keySet(img_uml$Class), sub : keySet(img_uml$Class) "
  + "with sub (<--{IsSuperClassOf} | <--{IsInterfaceOfClass}) "
  + "         <--{IsTypeDefinitionOf} super "
  + "reportSet sub, super end",
  "from t : $  reportMap t, nthElement(t, 0) end",
  "from t : $$ reportMap t, nthElement(t, 1) end");
```

The name of the new edge type to be created in the target schema is uml.IsA. The allowed source and target node type is uml.Class, which is referenced by the variable umlClass. The first GReQL query results in a set of tuples of the form (subtype, supertype). For each of those tuples, an IsA edge is created in the target graph. The next query returns a map from edge archetype to the archetype of the start node. With those informations, the transformation framework determines the image of the start node, and this acts as start node of the new IsA instance in the target graph. The same is done for the end node.

Extracting relevant parts of the legacy system. Now, that the parts of the legacy system which should be migrated into a service are identified, and it is known how the relevant classes interact with each other, a decision has to be made on *how* the migration should be done. Typically, there are at least two choices: (a) the system as a whole is left as-is, and a wrapper layer, which translates between messages specified in the service specification and corresponding method calls in the legacy code is created, or (b) the functionality needed is extracted from the legacy system leaving out the irrelevant parts. Since we are interested only in the resource managing feature, but do not want to use other features nor the GUI, we apply the second approach.

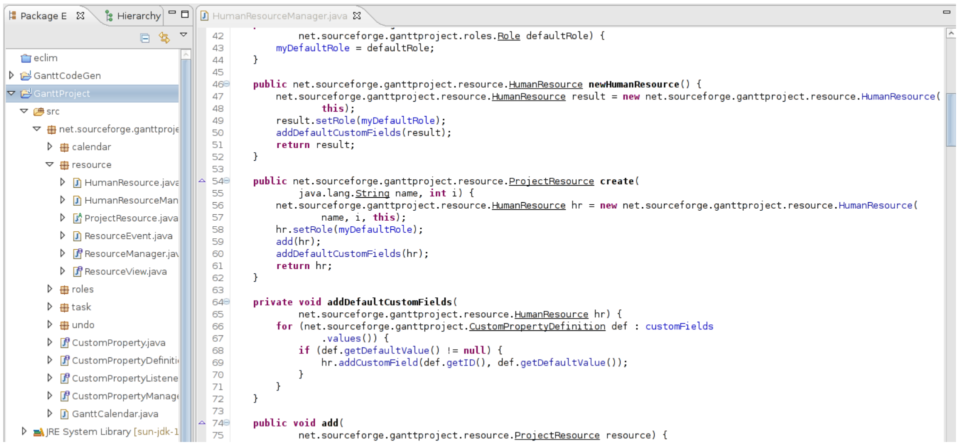
The GReQL query in Listing 8 collects all dependencies of the HumanResourceManager, which are required to form an executable and coherent target service.

All in all, some more than 20 classes and interfaces are found. In this example, the legacy programming language and the target language are both Java. Thus, the Java code generator was applied instead of writing a customized transformation. Given a TGraph conforming the Java schema and the set of nodes which should be considered (the GReQL query's result), it serializes the relevant parts back into Java code. The result is shown in Figure 5.

What remains left is the adaption of the extracted code to the service specification. How existing tools provided in the SOMA toolset can be used to simplify this task is depicted in [FWGH09].

```
from hrmClass: V{ClassDefinition}, hrmMethod: V{MethodDefinition}, usedType: V{Type}
with hrmClass.name = "HumanResourceManager"
  and hrmClass <--{IsClassBlockOf} <--{IsMemberOf} hrmMethod
  and (hrmMethod (
        (// method invocations
         <--{IsBodyOfMethod} <--{IsStatementOfBody}
         (<--{AttributedEdge,^IsBreakTargetOf,^IsContinueTargetOf,^IsTypeDefinitionOf})*
         <--{IsDeclarationOfInvokedMethod} -->{IsMemberOf} -->{IsClassBlockOf}
        ) | ( // method parameters of type usedType
         <--{IsParameterOfMethod} <--{IsTypeOf}+ <--{IsTypeDefinitionOf}
        ) | ( // hrmMethod uses usedType by declaring a variable of this type
         <--{IsBodyOfMethod} <--{IsStatementOfBody}
         (<--{AttributedEdge,^IsBreakTargetOf,^IsContinueTargetOf,^IsTypeDefinitionOf})*
         <--{IsTypeOfVariable} <--{IsTypeDefinitionOf}
        ) | ( // hrmMethod has usedType as return type
         <--{IsReturnTypeOf} <--{IsTypeDefinitionOf}
        ) ) usedType
    or hrmClass (
        (// hrmClass has a field of type usedType
         <--{IsClassBlockOf}<--{IsMemberOf}<--{IsFieldCreationOf}<--{IsTypeOfVariable}
         <--{IsTypeDefinitionOf}
        ) | ( // hrmClass derives from or implements usedType
         (<--{IsSuperClassOfClass} | <--{IsInterfaceOfClass}) <--{IsTypeDefinitionOf}
        ) | ( // hrmClass has a member class defining usedType (transitive)
         (<--{IsClassBlockOf} <--{IsMemberOf})+
        ) ) usedType )
reportSet usedType end
```

Listing 8: The GReQL query for retrieving dependencies



```
            net.sourceforge.ganttproject.roles.Role defaultRole) {
        myDefaultRole = defaultRole;
    }

    public net.sourceforge.ganttproject.resource.HumanResource newHumanResource() {
        net.sourceforge.ganttproject.resource.HumanResource result = new net.sourceforge.ganttproject.resource.HumanResource(
                this);
        result.setRole(myDefaultRole);
        addDefaultCustomFields(result);
        return result;
    }

    public net.sourceforge.ganttproject.resource.ProjectResource create(
            java.lang.String name, int i) {
        net.sourceforge.ganttproject.resource.HumanResource hr = new net.sourceforge.ganttproject.resource.HumanResource(
                name, i, this);
        hr.setRole(myDefaultRole);
        add(hr);
        addDefaultCustomFields(hr);
        return hr;
    }

    private void addDefaultCustomFields(
            net.sourceforge.ganttproject.resource.HumanResource hr) {
        for (net.sourceforge.ganttproject.CustomPropertyDefinition def : customFields
                .values()) {
            if (def.getDefaultValue() != null) {
                hr.addCustomField(def.getID(), def.getDefaultValue());
            }
        }
    }

    public void add(
            net.sourceforge.ganttproject.resource.ProjectResource resource) {
```

Figure 5: The generated code of the HRM and its dependencies

6 Scalability, Performance and Expressiveness

The underlying TGraph data structures have proven to scale well in different real-life projects, where graphs containing several millions of nodes and edges are handled. The slight overhead of having edges as first class objects is compensated by the ability to formulate efficient queries and transformations.

When comparing GReQL queries like the dependencies query in Listing 8 to tools like JDepend or Eclipse's "References" feature, the performance is almost equal, but the flexibility is much bigger. GReQL allows querying any informations from graphs conforming to any schema, and it allows formulating and executing ad-hoc queries.

The GReTL transformation operations presented in this paper are the most fine-granular means to specify transformations, and they provide the basic building blocks of the transformation framework. Unfortunately, their expressiveness is quite low. Therefore, research on and implementation of higher order transformations has been started. Those will make GReTL transformations more applicable and concise. For example, there is a generic copy transformation that can be used when parts of the source and target graph should be equal, or the ability to specify chains of transformations, where the target graph of transformation n serves as source graph for the transformation $n+1$. Additionally, means for specifying transformations visually are in the works.

7 Related Work

In the reengineering world, model-driven approaches are widely used. The central document for most reengineering techniques is the source code of the systems under analysis. The underlying data structures used for representing models vary among relational databases, object-networks, or graphs. Graphs have shown their applicability as general data structure for reverse engineering [HSEW06]. Due to the the explicit notion of edges as first class citizens, the TGraph approach provides performant querying and transformations.

A basic outline on model-driven approaches is given by OMG's Model-Driven Architecture [OMG03]. The OMG initiative Architecture Driven Modernization ([KU07], [Khu08]) tries to create standards, whose application enables model-driven modernization in larger industrial contexts, grounded on QVT transformations.

Concerning model transformation languages, the current state-of-the-art is set by the OMG Query/View/Transformation specification [OMG08]. Its central part is the Relations language, which offers fully declarative, bidirectional transformations. Contrasting MDA-style transformation languages like QVT or ATL [ATL09], there are also languages based on term systems and grammars, like Stratego [Vis01] and TXL [Cor04], which are most applicable if transformations should produce code from code.

Query- and transformation techniques used in this work (GReQL and GReTL) are directly based on TGraph technology, which has shown its applicability to both visual models [ESU97] and code in reengineering activities [ERW08].

Many publications deal with SOAs in general, and how to introduce them in industrial contexts. Popular examples are IBM's SOMA method [AGA+08] or Bell's Service-Oriented Modeling [Bel08]. The question how to migrate legacy assets into the new infrastructure is still to be solved, and only few articles on this topic can be found. A short introduction into challenges and experiences is given in [Gim07]. In [FWGH09], an overview on how the SOMA method can be extended with graph-based query and transformation techniques is given. The planning of SOA migration projects is enlightened by the SMART approach [Smi07].

In [Mat08] Matos describes a methodology for graph- and transformation-based migration into SOAs, which shows some similarities to the soamig project. Fleurey et al. present another model-driven migration approach which was successfully applied in a larger industrial context in [FBB$^+$07]. Yet another MDA-migration approach is depicted by Correia et al. in [CMHER07]. As representation of source code, they also use graphs, but before creating this representation, the code is specifically annotated according to which legacy parts contribute to which part of the target architecture. These annotations are used in the transformations. In contrast, in the soamig project we try to locate and extract possible service candidates directly without an intermediate step. A tool-driven stepwise migration approach is presented by Marchetto and Ricca in [MR08]. In a case study, a simple Java application was migrated to a equivalent webservice. However, this approach mostly deals with wrapping relevant functionality in the legacy code with adaptors, to enable their usage as webservice. In contrast, the goal of the soamig project is to supersede the legacy by transforming self-contained parts of its functionality into services.

8 Conclusion and Future Work

The work presented here provides a first step towards a coherent model-driven approach to software migration by combining graph-based reengineering techniques with MDA-style transformations.

In an initial case-study, a possible service candidate was identified in the project planning tool GanttProject. Some architecture recovery was performed using a GReTL transformation. To extract the legacy parts, the GReQL graph query language was used to retrieve all dependencies that have to be migrated to keep the code functional. Finally, the actual technical migration was performed by using a code generator to serialize all the needed nodes and edges from the Java TGraph representation back into source code.

The soamig project tries to take all different kinds of artifacts into account that can be useful during a migration project, instead of focussing solely on code. Work on an integrated metamodel encompassing source code (Java and Cobol), architecture, and business processes has just been started. Models conforming this metamodel will give a consolidated view on all relevant parts of the system, and make them subject to querying and transformations. Relations between code, architecture and business processes found out in reengineering activities will be manifested using traceability links between the elements. It is intended to do this linking at least semi-automatically, for example by using traces gathered with AspectJ instrumentation while executing (parts of) one specific business process with the legacy application.

The results of the research performed in the soamig project will be evaluated on larger system provided by our industrial partner.

References

[AGA+08] A. Arsanjani, S. Ghosh, A. Allam, T. Abdollah, S. Ganapathy, and K. Holley. SOMA: A method for Developing Service-Oriented Solutions. *IBM Systems Journal*, 47(3):377–396, 2008.

[ATL09] ATLAS Group. *ATL: User Guide*, 2009.

[BE08] D. Bildhauer and J. Ebert. Querying Software Abstraction Graphs. In *Working Session on Query Technologies and Applications for Program Comprehension (QTAPC 2008), collocated with ICPC 2008*, 2008.

[Bel08] M. Bell. *Service-Oriented Modeling: Service Analysis, Design, and Architecture*. John Wiley & Sons, 2008.

[BV09] A. Baldauf and N. Vika. Java-Faktenextraktor für Gupro. Studienarbeit, University Koblenz-Landau, Institute for Software Technology, 2009.

[CMHER07] R. Correia, C. Matos, R. Heckel, and M. El-Ramly. Architecture Migration Driven by Code Categorization. In *Software Architecture, First European Conference, ECSA, Aranjuez, Spain, September 24-26, Proceedings*, volume 4758 of *Lecture Notes in Computer Science*. Springer, 2007.

[Cor04] J. R. Cordy. TXL: A Language for Programming Language Tools and Applications. In *Proc. LDTA, ACM 4th International Workshop on Language Descriptions, Tools and Applications Barcelona*, pages 1–27, 2004.

[EH09] J. Ebert and T. Horn. The GReTL Transformation Language. int. Report, 2009.

[EKRW02] J. Ebert, B. Kullbach, V. Riediger, and A. Winter. GUPRO. Generic Understanding of Programs - An Overview. *Electronic Notes in Theoretical Computer Science*, 72(2), 2002.

[ERW08] J. Ebert, V. Riediger, and A. Winter. Graph Technology in Reverse Engineering, The TGraph Approach. In R. Gimnich, U. Kaiser, J. Quante, and A. Winter, editors, *10th Workshop Software Reengineering (WSR 2008)*, volume 126 of *GI Lecture Notes in Informatics*, pages 67–81. GI, 2008.

[ESU97] J. Ebert, R. Süttenbach, and I. Uhe. Meta-CASE in Practice: a Case for KOGGE. In *Advanced Information Systems Engineering, 9th international Conference, CAiSE'97*, volume 1250 of *LNCS*, pages 203–216. Springer, 1997.

[FBB+07] F. Fleurey, E. Breton, B. Baudry, A. Nicolas, and J.-M. Jezequel. Model-driven Engineering for Software Migration in a Large Industrial Context. In *Model Driven Engineering Languages and Systems: 10th international conference, MODELS, Nashville, USA; proceedings*, volume 4735, pages 482–497, Berlin, 2007. Springer.

[FWGH09] A. Fuhr, A. Winter, R. Gimnich, and T. Horn. Extending SOMA for Model-Driven Software Migration into SOA. In *11. Workshop Software-Reengineering, Bad Honnef, 4.-6. Mai 2009*, 2009.

[Gim07] R. Gimnich. SOA Migration: Approaches and Experience. *Softwaretechnik-Trends*, 27(1), 2007.

[HSEW06] R. C. Holt, A. Schürr, S. Elliott Sim, and A. Winter. GXL: A Graph-Based Standard Exchange Format for Reengineering. *Science of Computer Programming*, 60(2):149–170, April 2006.

[Khu08] V. Khusidman. ADM Transformation, 2008. `http://www.omg.org/cgi-bin/apps/doc?admtf/08-06-10.pdf`.

[KU07] V. Khusidman and W. Ulrich. Architecture-Driven Modernization: Transforming the Enterprise, 2007. `http://www.omg.org/cgi-bin/apps/doc?admtf/07-12-01.pdf`.

[KW98] B. Kullbach and A. Winter. Querying as an Enabling Technology in Software Reengineering. In *Proceedings of the 3nd European Conference on Software Maintenance and Reengineering*, pages 42–50. IEEE Computer Society, 1998.

[Mat08] C. Matos. Service Extraction from Legacy Systems. In D. Hutchison, H. Ehrig, R. Heckel, T. Kanade, and J. Kittler, editors, *Graph Transformations: 4th International Conference, ICGT, Leicester, United Kingdom; proceedings*, volume 5214, pages 505–507, Berlin, Heidelberg, 2008. Springer-Verlag.

[MR08] A Marchetto and F Ricca. Transforming a Java application in a equivalent Web-services based application: Toward a Tool Supported Stepwise Approach. In *Proceedings Tenth IEEE International Symposium on Web Site Evolution, Beijing, China (WSE)*. IEEE Computer Society, 2008.

[OAS06] OASIS. *Reference Model for Service Oriented Architecture 1.0*, October 2006. `http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf`.

[OMG03] OMG. MDA Guide Version 1.0.1, 2003. `http://www.omg.org/docs/omg/03-06-01.pdf`.

[OMG08] The Object Management Group. Meta Object Facility (MOF) 2.0: Query/View/Transformation Specification v1.0, 2008.

[Sch09] H. Schwarz. Towards a Comprehensive Traceability Approach in the Context of Software Maintenance. In *13th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, 2009.

[SERW08] H. Schwarz, J. Ebert, V. Riediger, and A. Winter. Towards Querying of Traceability Information in the Context of Software Evolution. In R. Gimnich, U. Kaiser, J. Quante, and A. Winter, editors, *10th Workshop Software Reengineering (WSR 2008)*, volume 126 of *GI Lecture Notes in Informatics*, pages 144–148, Bonn, 2008.

[SL06] A. Siffring and J. Lind. crossvision Legacy Integrator – Creating New Services and Value from Existing Systems, 2006.

[Smi07] D. B. Smith. Migration of Legacy Assets to Service-Oriented Architecture Environments. In *Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 174–175. IEEE Computer Society, 2007.

[Sne97] H. M. Sneed. Metriken für die Wiederverwendbarkeit von Softwaresystemen. *Informatik Spektrum*, 6, 1997.

[Vis01] E. Visser. Stratego: A Language for Program Transformation based on Rewriting Strategies. System Description of Stratego 0.5. In A Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01), LNCS 2051, Berlin*, pages 357–361, 2001.