GraX – An Interchange Format for Reengineering Tools*

Jürgen Ebert Bernt Kullbach Andreas Winter
University of Koblenz-Landau
Institute for Software Technology
Rheinau 1, D-56075 Koblenz, Germany
(ebert|kullbach|winter)@uni-koblenz.de

Abstract

Current research in software reengineering offers a great amount of tools specialized on certain reengineering tasks. The definition of a powerful common interchange format is a key issue to provide interoperability between these tools. This paper discusses aspects of data interchange formats for exchanging reengineering related data. It proposes a graph-based format to exchange both application specific concepts and data by XML documents.

Keywords: reengineering, data interchange format, tool interoperability, graph technology

1. Motivation

An important topic at WCRE'98 were talks [29, 30] and discussions on representation formats used in different reengineering toolsets. As a result of these discussions a general and powerful format allowing the exchange of reengineering data between different toolsets was required. This paper aims at motivating and presenting an approach to such an interchange format in order to continue the discussions from WCRE'98.

Current activities in software reengineering research mostly focus only on *isolated problems* of representing or analyzing software systems. Research addresses source code extraction [53], architecture recovery [23, 28, 31], concept analysis [45], data flow analysis [1, 37], pointer analysis [2, 48], program slicing [32, 50, 56], query techniques [33, 36, 39], source code visualization [38, 49], object recovery [7, 22, 35, 44], restructuring [52], or remodularization [43, 57].

All of these approaches give well elaborated support to certain aspects in software reengineering but unfortunately they only focus on their specific view to software reengineering. However, there is a need for a common reengineering toolset combining the variety of reengineering tasks into one single powerful workbench. Of course, due to complexity in the reengineering domain, it is almost impossible to develop such an "all inclusive" toolset alone. On the contrary it may be useful to combine the existing toolsets into an integrated reengineering workbench. In order to do so a suitable *interchange format* for exchanging the data between these tools has to be defined and realized. This format should be *general enough* to express application specific data. But it should also be *concrete enough* to be processable and interpretable by different tools.

In the following we propose *GraX* (Graph eXchange format) as a reengineering interchange format. *GraX* is formally based on *TGraphs* which define a very general class of graphs. In *GraX* XML is used as a means of notation.

This paper is organized as follows: Section 2 works out aspects which have to be recognized when defining reengineering interchange formats. Section 3 introduces *TGraphs* as a general means for representing reengineering data along with small examples from the *GUPRO* [15] project.[1] The *EER/GRAL* approach on graph-based conceptual modeling is presented in section 4 including its use in describing reengineering specific domain knowledge. The concrete *GraX* syntax is introduced in section 5. The paper ends with a conclusion and outlook in section 6.

2. Aspects of interchange formats

The definition of a general interchange format can be viewed from a conceptual as well as from a syntactical point of view. In this section requirements for interchange formats are derived from these aspects and related work is classified.

[1] Information on *GUPRO* including the technical reports cited in this paper is available from `http://www.gupro.de/`.

2.1. Conceptual aspect

With respect to concrete reengineering tasks different views to software systems become relevant. On the one hand the representation of software systems is affected by the *programming languages* used for implementing these software systems. Exchange formats for reengineering tools have to represent source code information for different single language systems (e. g. in C [6, 39] or Cobol [26]) and even in multi-language systems (e. g. [34]). On the other hand reengineering tools cope with various *levels of abstraction* [29]. These range from very fine-grained source code representations (e. g. in the fields of detailed data flow and control flow analysis) to coarse-grained source code representations (e. g. in the field of architectural understanding or in detecting structural source code dependencies). As a consequence, a reengineering interchange format has to encounter various kinds of data describing specific aspects of reengineering tasks with respect to language and abstraction level aspects.

Experiences from other areas e. g. in the work on interoperability of requirements engineering tools have shown that due to the heterogeneity of the subject domain no common meta model can be provided as a basis for data exchange [18]. If we suppose that such a common conceptual model can also not be provided in the domain of reengineering tools, a common interchange format has also to incorporate *schema*-like domain specific knowledge about the data to be exchanged.

Therefore a common interchange format has to support the exchange of *schema* and *instance* data.

2.2. Syntactical aspect

To define a general interchange format we have to fix the notation used for exchanging schema and instance information. Here we have to agree on the kind of *abstract syntax* for describing the mathematical structures underlying the interchange format and the *concrete syntax* for noting down the information to be interchanged.

Looking at the kind of abstract syntax used in reengineering approaches for internal source code representation one can identify syntax tree-based approaches [8], relational and algebraic approaches, [11, 31, 39], graph-based approaches [7, 15, 34, 38], and concept lattice approaches [45]. Domain specific schema information is explicitly stored in generic reengineering tools as RIGI [38], PBS [20], or GUPRO [15]. For exporting these internal data structures proprietary textual ASCII notations like RSF [58], TA [25], or *.g* [10] are used.

A general data exchange approach should offer an easily implementable and extensible format which is efficient in time and space. To avoid proprietary notations it should be based on an open standard.

2.3. Related work on reengineering interchange formats

The idea of an interchange format for data exchange between related tools is not new. There have e. g. been efforts like STEP in the domain of product data interchange [27] or CDIF in the domain of CASE tool interoperability [5]. CDIF proposes a family of standards for exchanging models with respect to the various description paradigms in the software design, (e.g. data models, data flow models, state event models). The respective standards consist of a meta model describing the concepts of the data to be interchanged. Concrete CASE models are interchanged by an ASCII language which is compliant to the meta model.

Interoperability of reengineering tools can be obtained by defining suitable tool frameworks. Woods et al. [59] propose CORUM as a framework addressing the integration of program understanding tools that operate on the code level of abstraction. In CORUM II [29] this approach was extended to cover the architectural level of abstraction as well. Koschke et al. [30] propose an interchange framework covering both levels.

Almost every intermediate representation used in reengineering tools that offers sufficient modeling power (e. g. ASFIX [51]) may serve as a candidate for a common data interchange format. In addition to their abstract notation these intermediate representations have to be completed with an export format defining the concrete syntax. Classifications of those portable intermediate representations are given in [30, 41].

There are also formats which are especially used or developed for interchanging reengineering data.

A popular exchange format is defined in Rigi [38]. Here *Rigi Standard Format (RSF)* [58] is used for importing and exporting data. Many research groups have used Rigi for visualization purposes (e. g. [17, 40]). RSF represents typed, attributed, relational graphs i. e. the abstract syntax of RSF is a graph. The concrete syntax of RSF is provided by a tuple language. RSF is based on an explicitly defined schema that identifies vertex types, edge types, attributes, and the colors used for visualization. In this schema, vertex types are just listed, while edge types, attributes and colors are provided as tuples. Higher conceptual modeling concepts like generalization hierarchies on vertex and edge types are not supported in RSF.

The *Tuple-Attribute Language (TA)* [25] can be viewed as an extension to RSF. Like RSF the abstract notation is restricted to relational graphs i. e. two vertices may not be connected by two edges of the same type. The concrete syntax in TA represents graphs through tuple and attribute sublanguages. Schema and data information is provided using this tuple representation. TA gives notational support for describing inheritance relations on vertex and edge types

including multiple inheritance. In [4] TA is proposed as a possible interchange format in connecting architecture level frameworks.

The *GraX* interchange format, which is proposed in this paper, is founded on a more general graph structure. The abstract syntax of this exchange format is defined by directed, typed, attributed, and ordered graphs (*TGraphs*) which allow multiple edges and loops. Application specific schema information is modeled by graph-based conceptual modeling techniques providing contemporary modeling power. In *GraX*, schemas are provided as *TGraphs* as well. The concrete exchange syntax of *GraX* is based on XML (Extensible Markup Language) [54].

3. Modeling with *TGraphs*

We propose graphs as a means for interchanging data between reengineering tools. In addition to their modeling power, graphs are a well-understood mathematical concept (e. g. [24]) which defines a powerful abstract data type with a great amount of experienced algorithms (e. g. [3, 12, 19]). According to the syntactical aspects of interchange formats graphs are a good basis for the *abstract syntax* for representing information in reengineering tools.

Graphs are a common representation in the reengineering domain. Some tools are directly founded on graph- or tree-based representations. Others, e. g. relation-based approaches, can easily be transformed into graph-like structures. The class of graphs used for interchange must be *as rich as possible* to be able to express as much structure as needed. But it should also be *scalable* to only cover those structural aspects that are needed.

Such a *common graph model* is given by *TGraphs* [13, 16]. *TGraphs* are *directed* graphs, whose vertices and edges may be *attributed* and *typed*. Each type can be assigned an individual attribute schema specifying the possible attributes of vertices and edges. The type system allows multiple inheritance. Furthermore, *TGraphs* are *ordered*, i. e. the vertex set, the edge set, and the sets of edges incident to a vertex have a total ordering. This ordering gives modeling power to describe sequences of objects (e. g. parameter lists) and facilitates the implementation of deterministic graph algorithms.

TGraphs are more general than other graph models e. g. conceptual graphs [46] or PROGRES graphs [42]. Conceptual graphs define a subclass of *TGraphs* which is restricted to bipartite and connected graphs while PROGRES graphs do not support attributed edges and ordering of graph elements. *TGraphs* are also superior to object-based representations [55], because edges are *first class entities* with their own types and attributes. As a consequence, edges can be treated independently from vertices which e. g. allows traversing edges in both directions.

In the context of practical applications, not all properties of *TGraphs* have to be used to their full extent. E. g. in the case of modeling abstract syntax trees *TGraphs* can be restricted to tree-like graphs. Other applications may require DAG-like, undirected or relational graphs. All these graphs can be expressed by restricting *TGraphs* accordingly.

Summarizing, *TGraphs* are an *expressive* abstract means for representing all or at least most of the data structures used in reengineering tools which is *scalable* with respect to the application context.

3.1. Representing instance information through *TGraphs*

Intermediate source code representations in program understanding and renovation tools deal with a wide spectrum of abstraction levels ranging from fine-grained structures on the *code level* to coarse-grained structures on the *architectural level*. In the following both ends of this spectrum will be addressed by giving two representational examples. In each example different classes of *TGraphs* representing the application specific knowledge are used.

Fine-grained modeling of programs (code level)

On the code level of abstraction the analysis may concentrate on statements, expressions, variables, operands, and the contains structure of statements. These objects and relationships have to be represented in an interchange structure between parsing components and analyzing components or between different analyzing components. Such a fine-grained analysis of the code fragment in figure 1 may be based on the *TGraph* in figure 2.

```
while x > 0 do
    repeat
        y := y + 1
    until (y = x);
    x := x - 1
od
```

Figure 1. Source code fragment

Such a piece of program text is usually transformed into an abstract syntax tree (AST). If every identifiable object is represented by exactly one vertex and every occurrence of an object is represented exactly once by an edge, this leads to the DAG-like structure in figure 2. *Variable x* is modeled by exactly one vertex ($v3$) having four outgoing edges representing uses (*isOperandIn*-edges $e4$, $e9$ and $e15$) and definitions (*isDefinedBy*-edge $e12$) of variable *x* in the order they occur. Details of *Operators*, *Constants* and *Variables* are expressed by vertex attributes *id* or *value*.

The fine-grained modeling on code level in figure 2 is done by a *TGraph* which is directed, vertex-attributed, vertex- and edge-typed, ordered and acyclic.

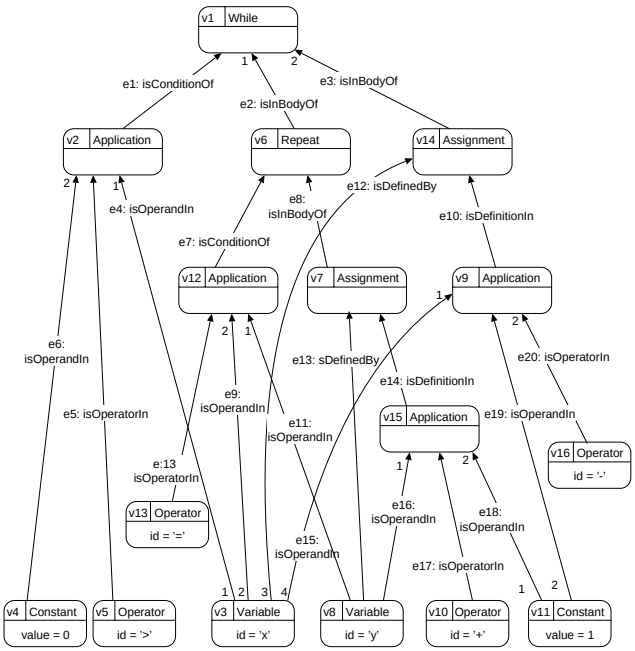

Figure 2. Fine-grained program graph

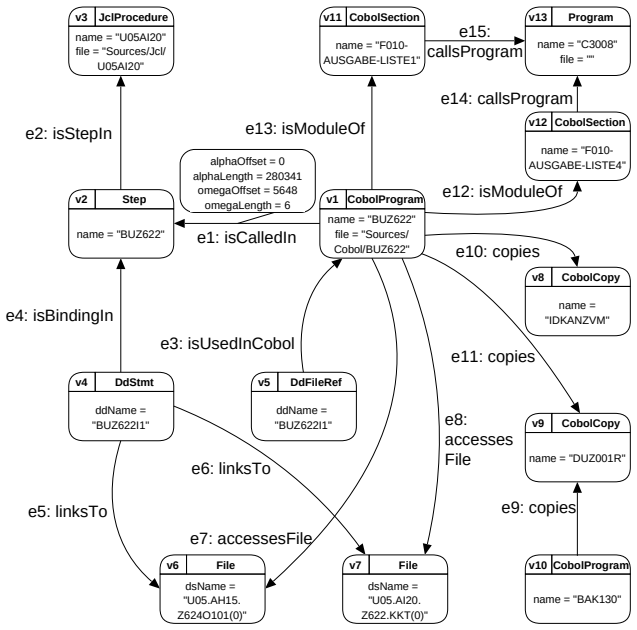

Figure 3. Multi-language system (extract)

Coarse-grained modeling of multi-language systems (architectural level)

On the architectural level of abstraction, analysis may concentrate on the relationships between system components in different programming languages. The *TGraph* in figure 3 shows an extract of the software system of a large insurance company [34]. Due to complexity it is restricted only to some concepts that are related to JCL, Cobol and their interdependencies.

The *TGraph* shows parts of the embedding of *CobolProgram* BUZ622 ($v1$) into the whole system. It is called ($e1$) by *Step* BUZ622 ($v2$), accesses ($e7$,$e8$) two *Files* ($v6$, $v7$), includes ($e10$, $e11$) two *CobolCopies* ($v8$, $v9$) and contains ($e12$, $e13$) two *CobolSections* ($v11$, $v12$) which call ($e14$, $e15$) an external *Program* ($v13$). All vertices in this example are attributed with names (*name*, *ddname*, *dsname*) of the modeled source code artifacts. Furthermore, all edges carry coordinate attributes that link to the concrete positions in the source code. This position information is depicted in figure 3 only for edge $e1$ as an example. It is used together with the *file* attributes of *JclProcedures* and *CobolPrograms* to visualize the associated source code fragments in the *GUPRO* source code browser.

In this example a directed, vertex- and edge attributed, vertex- and edge-typed *TGraph* is used.

3.2. Formalization of *TGraphs*

The two *TGraph* examples in figure 2 and 3 indicate that *TGraphs* are a scalable means for representing source code in various reengineering applications on different levels of abstraction. Before proposing *TGraphs* as *abstract syntax* for a general reengineering interchange format the formal foundation has to be specified. In [13, 16] *TGraphs* are introduced as a mathematical structure using the $\mathcal{Z}$ specification language [47].

The basic elements (*Element*) of *TGraphs* are vertices (*Vertex*) and edges (*Edge*), which are identified through natural numbers. An edge may occur as incoming or outgoing, which is represented in the *Dir* attribute.

$$
\begin{aligned}
&\mathit{Element} ::= \mathit{vertex}\langle\langle \mathbb{N} \rangle\rangle \mid \mathit{edge}\langle\langle \mathbb{N} \rangle\rangle \\
&\mathit{Vertex} == \operatorname{ran} \mathit{vertex} \\
&\mathit{Edge} == \operatorname{ran} \mathit{edge} \\
&\mathit{Dir} ::= \mathit{in} \mid \mathit{out}
\end{aligned}
$$

TGraph elements may be associated with a type and with attribute value pairs. Type identifiers (*TypeId*) and attribute identifiers (*AttrId*) are derived from a given set *Id*. The association between attribute identifiers and their values is defined by a finite partial function *AttributInstanceSet*.

$$
\begin{aligned}
&[\mathit{Id}, \mathit{Value}] \\
&\mathit{TypeId} == \mathit{Id} \\
&\mathit{AttrId} == \mathit{Id} \\
&\mathit{AttributeInstanceSet} == \mathit{AttrId} \pfun \mathit{Value}
\end{aligned}
$$

Based on these definitions, *TGraphs* are specified by the $\mathcal{Z}$ schema in figure 4. *TGraphs* consist of finite and injective sequences of vertices (V) and edges (E), respectively. An incidence function Λ associates to each vertex the sequence of its incident edges together with their direction information. Types and attributes of graph elements are given

by the *type* and *value* functions. Further predicates ensure, that the incidence lists are injective [p1], that every edge occurs in exactly one incidence list as outgoing and in exactly one incidence list as incoming [p2], and that type and attribute functions are restricted to existing graph elements [p3], [p4]. This $\mathcal{Z}$ schema gives the formal foundation for the definition of the concrete *GraX* notation for interchanging *TGraphs* in section 5.

TGraph

V : iseq *Vertex*
E : iseq *Edge*
Λ : *Vertex* $\twoheadrightarrow$ seq(*Edge* $\times$ *Dir*)
type : *Element* $\twoheadrightarrow$ *TypeId*
value : *Element* $\twoheadrightarrow$ *AttributeInstanceSet*

$\Lambda \in \operatorname{ran} V \twoheadrightarrow \operatorname{iseq}(E \times Dir)$ [p1]

$\forall e : \operatorname{ran} E \bullet \exists_1 v, w : \operatorname{ran} V \bullet$
$(e, in) \in \operatorname{ran}(\Lambda(v)) \wedge$
$(e, out) \in \operatorname{ran}(\Lambda(w))$ [p2]

$\operatorname{dom} type = V \cup E$ [p3]

$\operatorname{dom} value = V \cup E$ [p4]

Figure 4. $\mathcal{Z}$ schema *TGraph*

4. Conceptual modeling with *EER*

Being a plain structural means for describing, *TGraphs* have no meaning of their own. The meaning of *TGraphs* corresponds to the context in which they are exchanged. This application context determines which vertex and edge types, which attributes and which incidence relations are modeled. *Conceptual modeling techniques* are used to define classes of *TGraphs* representing this application related knowledge.

In order to provide the definition of *TGraph* classes on a contemporary semantic level, we use the *EER/GRAL* approach to graph-based modeling [16]. Classes of *TGraphs* are defined through extended entity relationship diagrams (*EER*) which may be annotated with additional restrictions in *GRAL* (Graph Specification Language) [21]. In *EER* diagrams *entity* types and *relationship* types are used to specify *vertex* types and *edge* types together with their attribute definitions and incidences. Multiple *generalization* is allowed for vertex and edge types. Further structural information can be modeled by using *aggregations*.

EER diagrams allow to describe the concepts of the software systems to be represented in reengineering tools. They can be viewed as *conceptual models* of the application domain determining the meaning of exchanged data.

4.1. Representing application-specific knowledge through *EER* models

As said before, conceptual *EER* schemas restrict the set of *TGraphs* to those graphs representing application related data. Each reengineering task needs its specific source code representation. E. g. the examples of analyzing fine-grained program structures or of inspecting coarse-grained source code interdependencies in section 3 require different application-specific conceptual models.

Fine-grained conceptual model (code level)

The *EER* model[2] in figure 5 defines a small conceptual model for fine-grained program understanding. It defines the concepts *Statement*, *Expression* and *Operator* and their connecting relationships. *Statements* are subdivided into *Assignments* and *Loops* which themselves are (disjointly) specialized into *While* or *Repeat* loops. The concepts *Variable*, *Constant*, and *Application* are generalized into the concept *Expression*. The subconcepts of *Assignment*, *Loop* and *Application* are modeled as aggregations. E. g. an *Application* consists of exactly one *Operator* and at least one *Expression*.

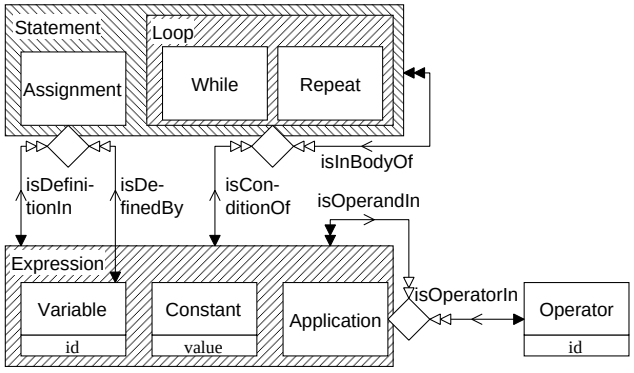

Figure 5. Fine grained conceptual model

The *TGraph* in figure 2 representing the program fragment in figure 1 is one possible instance of this conceptual model.

Multi-language conceptual model (architectural level)

On an architectural level the reengineer might be interested in the main building blocks such as *JclProcedures*, *Programs*, *CobolCopies* or *Files* and their interconnection.

[2] In the concrete notation of the *EER* dialect used for presentation, vertex types are represented by rectangles and edge types by (directed) arcs. Generalization is depicted by the usual triangle notation but also by graphically nesting object types. Within both notations an abstract generalization is symbolized by hatching. Aggregation is depicted by a diamond at the vertex type rectangle. Relationship cardinalities are given by an arrow notation at the participating vertex types.

The conceptual model in figure 6 depicts an extract of the multi-language conceptual model [34] related to the Cobol and JCL part. Here *CobolPrograms* may include *CobolCopies* and contain *CobolSections* which may call *Programs*. *CobolPrograms* are called by *Steps* which are collected in *JclProcedures*. By using *DdStmts* and *DdFileRefs* *CobolPrograms* access *Files*.

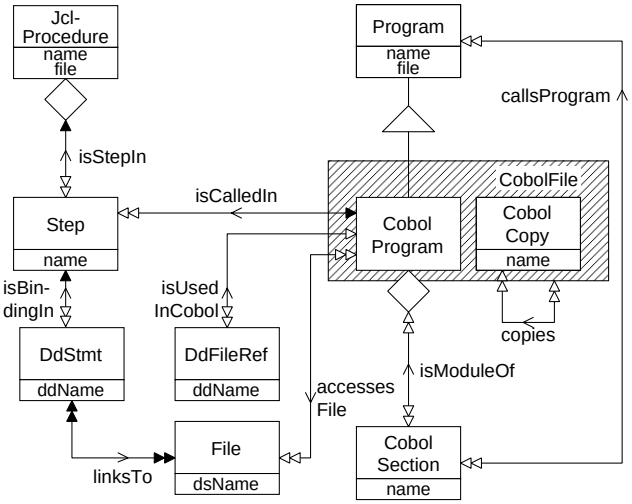

Figure 6. Multi-language conceptual model

This conceptual model defines the schema for *TGraphs* like the one in figure 3.

4.2. Formalization of *EER* models

The formal foundation of *EER* modeling of graph structures is defined in [9] by $\mathcal{Z}$ specifications. Each *EER* model denotes a set of corresponding *TGraphs* by describing valid vertex and edge types including their attribute and inheritance structures, the allowed connection between vertex types and edge types, and additional constraints (like degree restrictions).

4.3. Representing *EER* models through *TGraphs*

Since *EER* diagrams are structured information themselves, they may be described as *TGraphs* as well. The class of *EER TGraphs* can be defined by a *meta EER model* in such a way, that all the *TGraphs* representing an *EER* diagram are compliant to this meta schema [14]. This meta schema is given in figure 7.

Entity types are modeled in *EER TGraphs* as *EntityType* and relationship types as *RelationshipType* vertices. The incidences are modeled by *comesFrom* and *goesTo* edges. *Attribute* vertices representing attribute names can be associated by *hasAttribute* edges to *EntityType* and *RelationshipType* vertices. Attribute domains are specified using *Domain* vertices. Aggregation-like relationships are modeled

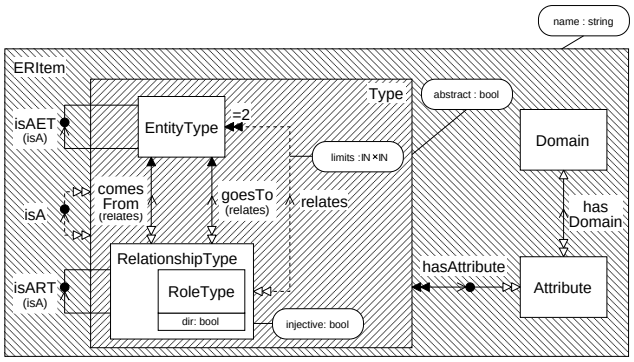

Figure 7. *EER* metaschema

by vertices of type *RoleType*. *isAET* edges and *isART* edges describe generalization hierarchies for entity types and relationship types.

According to the *EER* meta model in figure 7 the conceptual models in figure 5 and 6 can be represented by the *TGraphs* given in figures 8 and 9.

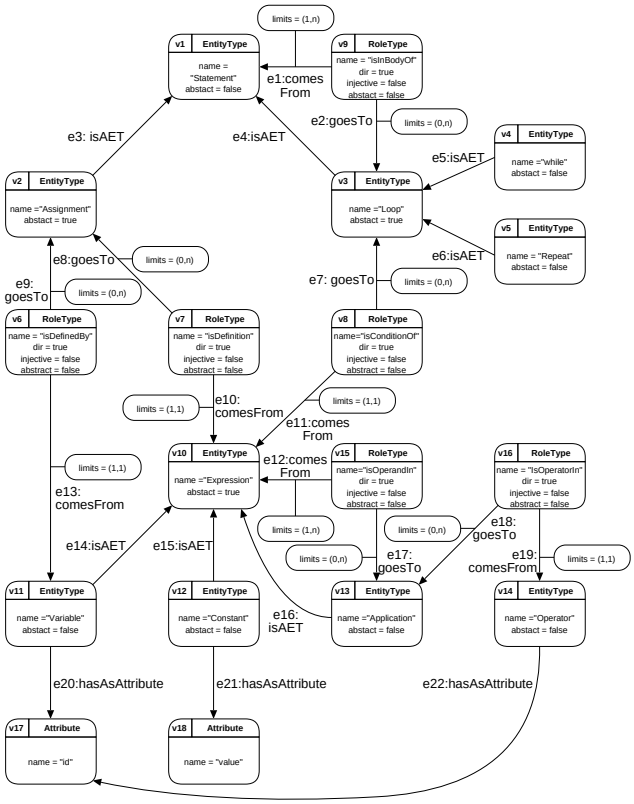

Figure 8. Fine-grained conceptual model as *EER TGraph*

The examples presented so far give evidence that *TGraphs* define an abstract syntax for representing instances and schemas for interchanging reengineering data. Because the *EER* meta model itself is an *EER* model it is representable and exchangeable as an *EER TGraph* as well.

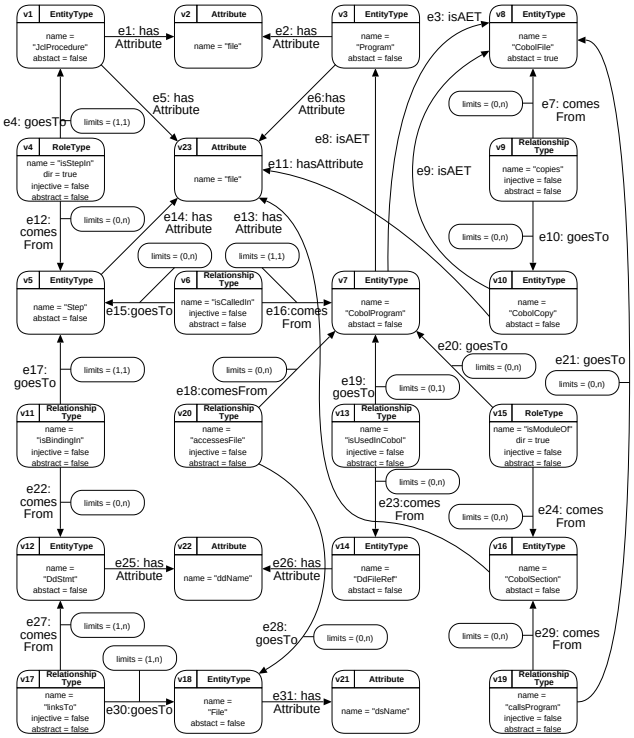

Figure 9. Multi-language conceptual model as *EER TGraph*

5. *GraX*

Having agreed on the abstract syntax of an exchange format, a concrete notation has to be fixed for *TGraphs*. It has to be ensured that translation between almost any internal representation and this format can be done easily. We have chosen XML [54] as a non proprietary interchange mechanism which fits to current internet technology.

XML offers a meta language for defining structures of documents in the world wide web. These structures are defined in document type definitions (DTD). Hereby the elements of documents including their attributes and consists-of relationships are specified. Concrete documents are described in a markup language according to the structures defined in their DTD. These DTDs enable a distributed and independent development of tools for visualization and analysis.

5.1. *GraX* document type definition

In a reengineering interchange format instance data have to be exchanged together with their conceptual models. Proceeding naively, every conceptual model can be translated into an appropriate DTD and the corresponding instance data is described in a suited XML document. Unfortunately this policy leads to *different* exchange formats for schemas and instances. As shown in section 4 schema and instance information can be based on the *same* abstract syntax. Thus in *GraX only TGraphs* have to be exchanged.

```
<!ELEMENT grax (vertex | edge)* >
<!ATTLIST grax
    schema CDATA    #REQUIRED >

<!ELEMENT vertex (attr)* >
<!ATTLIST vertex
    id     ID       #REQUIRED
    type   CDATA    #IMPLIED
    lambda IDREFS   #IMPLIED >

<!ELEMENT edge (attr)* >
<!ATTLIST edge
    id     ID       #REQUIRED
    type   CDATA    #IMPLIED
    alpha  IDREF    #REQUIRED
    omega  IDREF    #REQUIRED >

<!ELEMENT attr EMPTY >
<!ATTLIST attr
    name   CDATA    #REQUIRED
    value  CDATA    #REQUIRED >
```

Figure 10. XML document type definition for *TGraphs* (grax.dtd)

According to the formal specification of *TGraphs* in section 3.2 the *TGraph* document type definition is given in figure 10. A *GraX* document is attributed with its schema name and consists of vertex and edge elements. Both kinds of elements may contain attributes (attr) which consist of names and a values. Vertex and edge elements are identified by a required identifier id and both may be attributed with a type identifier. The ordering of vertices and edges is given by their textual position. Incidences including the orientation of edges are described as required alpha and omega attributes within edge elements. Furthermore, an optional attribute lambda can be associated with vertex elements describing the ordering of incident edges. Alpha, omega and lambda refer to the identifiers of vertex and edge elements.

XML does not support different name spaces for identifiers of different elements. So, in an additional constraint, lambda attributes have to be restricted to identifiers referencing edge elements, while alpha and omega attributes refer to vertex elements. For distinguishing vertices and edges we propose naming vertices beginning with ”v” and edges with ”e” followed by an integer. Attribute values in attr elements are of type string (CDATA). For notating concrete *GraX* documents suitable casting mechanisms have to be established for transferring values of other types into strings and vice versa.

5.2. Exchanging data using *GraX*

With respect to the *GraX* document type definition, the *TGraphs* representing reengineering related data can be exchanged by simple ASCII texts. A *GraX* document specify-

ing the multi-language system graph of figure 3 is given in figure 11.

```
1  <?xml version="1.0" ?>
2  <!DOCTYPE grax SYSTEM "grax.dtd" >
3  <grax schema = "http://www.gupro.de/schemas/multi.scx">
4        <vertex id = "v1" type = "CobolProgram" >
5              <attr name = "name" value = "BUZ622"/>
6        </vertex>
7  ...
8        <edge id = "e1" type = "isCalledIn"
9              alpha = "v1" omega = "v2" >
10             <attr name = "alphaOffset" value = "0"/>
11             <attr name = "alphaLength" value = "280341"/>
12             <attr name = "omegaOffset" value = "5648"/>
13             <attr name = "omegaLength" value = "6"/>
14       </edge>
15 ...
16 </grax>
```

Figure 11. Multi-language graph in *GraX* (extract)

GraX documents start with specifying the XML version and the underlying DTD in lines 1 and 2. This initial information is followed by the graph definition between the <grax> and </grax> tags which starts with the schema information (line 3). The CobolProgram vertex v1 is described as a vertex element in lines 4–6. Its name attribute with value BUZ622 is specified in the attr element in line 5. Analogously lines 8–14 show the edge e1 connecting v1 and v2 including its attributes.

Incidence lists describing the ordering of edges incident to a vertex are shown in figure 12 which is an extract of the *GraX* document describing the *TGraph* in figure 2. The vertex v3 is incident to the edges e4, e9, e12, and e15 in this order.

```
1  ...
2        <vertex id = "v3" type = "Variable"
3              lambda = "e4 e9 e12 e15" ></vertex>
4  ...
```

Figure 12. Incidence lists in *GraX*

A *TGraph* representing schema information has been shown in figure 6. Schema *TGraphs* like this can also be interchanged as *GraX* documents. A part of the *GraX* document describing this schema *TGraph* is given in figure 13.

These examples show that the *TGraph* document type definition provides a structure to describe instance and schema information related to reengineering information on different levels of abstractions. ASCII texts following this definition, of course, only describe *TGraphs* without checking if an instance *TGraph* matches its schema *TGraph*. But this can be done easily by a component for type checking instance graphs using the *GraX* interchange format.

```
1  ...
2        <vertex id = "v5" type = "entityType" >
3              <attr name = "name" value = "Step"/>
4              <attr name = "abstract" value = "false"/>
5        </vertex>
6        <vertex id = "v6" type = "RelationshipType" >
7              <attr name = "name" value = "isCalledIn"/>
8              <attr name = "injective" value = "false"/>
9              <attr name = "abstract" value = "false"/>
10       </vertex>
11       <vertex id = "v6" type = "EntityType" >
12             <attr name = "name" value = "CobolProgram"/>
13             <attr name = "abstract" value = "false"/>
14       </vertex>
15 ...
16       <edge id = "e15" type = "goesTo"
17             alpha = "v6" omega = "v7" >
18             <attr name = "limits" value = "(0,n)"/>
19       </edge>
20       <edge id = "e16" type = "comesFrom"
21             alpha = "v6" omega = "v5" >
22             <attr name = "limits" value = "(1,2)"/>
23       </edge>
24 ...
```

Figure 13. Multi-language conceptual model in *GraX* (extract)

6. Conclusion and future work

In this paper we have presented the *GraX* interchange format for exchanging reengineering related data. *GraX* offers a format which is both *general enough* to represent application specific knowledge and *concrete enough* to be processable and interpretable by different tools. The *abstract notation* of *GraX* is based on a general graph model and the *concrete notation* is based on the recommended open XML standard. In *GraX*, conceptual models representing application specific knowledge and instance data are exchanged by the *same* document type.

Criteria for the quality of interchange formats are proposed in [4] and [30]. In addition to its formal foundation *GraX* fulfills these criteria to a large extent.

Because *GraX* combines the exchange of instance data and their conceptual models *GraX* is not restricted to fixed application domains. By defining *EER* models *GraX* offers an *extensible interchange format*. Hereby *GraX* supports describing and exchanging reengineering data on *several levels of abstraction* independently from *programming languages*. Furthermore *mappings* between the exchange format and the represented sources can be defined by suitable conceptual models (e. g. the one in figure 6). Using XML, *GraX* documents are based on a *universal standard*. Unfortunately XML based languages blow up the *size* of textual descriptions. But translations into the *GraX* format produce representations which are linear in size to the length of the source code and standard compression tools can be used during transferring these documents. (Even so

it makes sense to use abbreviations for tags and attribute names in order to shorten *GraX* texts.) For internal representations as used in *GUPRO* or RIGI translations into this format and vice versa can be done in linear time with respect to the document size.

The interchange format proposed here is not only restricted to the reengineering domain. Like CDIF it can easily be applied to interchanging documents between CASE tools, as well. Both, the definition of concrete requirements engineering languages (conceptual models) and concrete documents (instances) can be exchanged between interoperable CASE tools and even between CASE and reengineering tools.

Future work has to be done on implementing further filters from various intermediate representations into *GraX* documents and vice versa. Due to the simplicity of the *GraX* DTD this seems to be an easy task from the *GraX* point of view. Further tools e. g. for checking the consistency of an instance document with respect to its conceptual schema can be realized based on the *GraX* DTD.

GraX covers the *conceptual* and the *syntactical aspects* as proposed in section 2. If one has agreed on the abstract and concrete syntax for interchanging instance and schema data, the remaining work will deal with the definition of conceptual models for certain application domains only.

Thus, to make reengineering tools interoperable, further work has to be done in agreeing on a set of application specific conceptual *reference models* and the description of their *meaning*.

Acknowledgement

We would like to thank Kostas Kontogiannis (Waterloo), Rainer Koschke (Stuttgart), Johannes Martin (Victoria) and Thomas Pühler (Koblenz) for valuable discussions on exchange formats which improved this work very much.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, Reading, 1986.

[2] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[3] C. Berge. *Graphs and Hypergraphs*, volume 6 of *North-Holland Mathematical Library*. North-Holland, Amsterdam, 2nd edition, 1976.

[4] I. Bowman, M. Godfrey, and R. Holt. Connecting Architecture Reconstruction Frameworks. In *Proceedings of the First International Symposium on Constructing Software Engineering Tools (CoSET'99)*, 1999.

[5] CDIF. Standardized CASE Interchange Meta-Model, EIA/IS-83. Technical report, Electronic Industries Association, Engineering Department, Washington D. C., July 1991.

[6] Y.-F. Chen, M. Y. Nishimoto, and C. V. Ramamoorthy. The C Information Abstraction System. *IEEE Transactions on Software Engineering*, 16(3):325–334, March 1990.

[7] K. Cremer. A Tool Supporting the Re-Design of Legacy Applications. In P. Nesi and F. Lehner, editors, *Proceedings of the 2nd Euromicro Conference on Software Maintenance & Reengineering*, pages 142–148, Los Alamitos, 1998. IEEE Computer Society.

[8] R. F. Crew. ASTLOG: A Language for Examining Abstract Syntax Trees. In *Proceedings of the Conference on Domain-specific Languages, October 15-17, 1997, Santa Barbara*. USENIX Association, Berkley, 1997.

[9] P. Dahm, J. Ebert, A. Franzke, M. Kamp, and A. Winter. TGraphen und EER-Schemata — Formale Grundlagen. In *[15]*, pages 51–65. 1998.

[10] P. Dahm and F. Widmann. Das Graphenlabor, Version 4.2. Fachbericht Informatik 11/98, Universität Koblenz-Landau, Institut für Informatik, Koblenz, 1998.

[11] A. Deursen and L. Moonen. Understanding COBOL Systems using Inferred Types. In *Proceedings of 7th International Workshop on Program Comprehension*. IEEE, Los Alamitos, 1999.

[12] J. Ebert. *Effiziente Graphenalgorithmen*. Akademische Verlagsgesellschaft, Wiesbaden, 1981.

[13] J. Ebert and A. Franzke. A Declarative Approach to Graph Based Modeling. In E. Mayr, G. Schmidt, and G. Tinhofer, editors, *Graph-theoretic Concepts in Computer Science*, volume 903 of *LNCS*, pages 38–50. Springer, Berlin, 1995.

[14] J. Ebert, A. Franzke, M. Kamp, D. Polock, and F. Widmann. TGREP – Graphklasse zur Repräsentation von TGraph–bezogenen Ausdrücken und Prädikaten. Projektbericht 12/97, Universität Koblenz-Landau, Institut für Softwaretechnik, Koblenz, 1997.

[15] J. Ebert, R. Gimnich, H. H. Stasch, and A. Winter, editors. *GUPRO — Generische Umgebung zum Programmverstehen*. Fölbach, Koblenz, 1998.

[16] J. Ebert, A. Winter, P. Dahm, A. Franzke, and R. Süttenbach. Graph Based Modeling and Implementation with EER/GRAL . In B. Thalheim, editor, *Conceptual Modeling — ER'96*, volume 1157 of *LNCS*, pages 163–178. Springer, Berlin, 1996.

[17] T. Eisenbarth, R. Koschke, E. Plödereder, G.-F. Girard, and M. Würthner. Projekt Bauhaus – Interaktive und inkrementelle Wiedergewinnung von SW-Architekturen. In J. Ebert and F. Lehner, editors, *Proceedings Workshop Reengineering, Bad Honnef, 27.-28. May 1999*. University of Koblenz-Landau, Koblenz, 1999.

[18] J. Ernst. *Introduction to CDIF*. http://www.eigroup.org/cdif/intro.html, Sept. 1997.

[19] S. Even. *Graph Algorithms*. Pitman, Maryland, 1979.

[20] P. J. Finnigan, R. C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. A. Müller, J. Mylopoulos, S. G. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, 1997.

[21] A. Franzke. GRAL: A Reference Manual. Fachbericht Informatik 3/97, Universität Koblenz-Landau, Fachbereich Informatik, Koblenz, 1997.

[22] H. Gall and R. Klösch. Finding Objects in Procedural Programs: An Alternative Approach. In L. Wills, P. Newcomb, and E. Chikofsky, editors, *Proceedings of the Second Working Conference on Reverse Engineering (WCRE). Toronto, Ontario, Canada, July, 14-16 1995*, pages 208–216, Los Alamitos, California, 1995. IEEE Computer Society Press.

[23] J.-F. Girard and R. Koschke. Finding Components in a Hierarchy of Modules - a Step towards Architectural Understanding. In *Proceedings of the International Conference on Software Maintenance 1997*, pages 58–65. IEEE Computer Society Press, 1997.

[24] F. Harary. *Graph Theory*. Addison-Wesley, Reading, 1969.

[25] R. Holt. An Introduction to TA: The Tuple-Attribute Language. http://www.turing.toronto.edu/∼holt/papers/ta.html, 1997.

[26] M. Hümmerich. Entwicklung und prototypische Implementation eines konzeptionellen Modelles zum Reverse-Engineering von ANSI85-COBOL-Programmen. Studienarbeit S 380, Universität Koblenz-Landau, Fachbereich Informatik, Koblenz, Juni 1995.

[27] ISO 10303 (STEP), 1994.

[28] R. Kazman and J. Carrière. Playing Detective: Reconstructing Software Architecture from Available Evidence. *Automated Software Engineering*, 6(2):107–138, April 1999.

[29] R. Kazman, S. Woods, and J. Carrière. Requirements for Intergrating Software Architecture and Reengineering Models: CORUM II. In *Fifth Working Conference on Reverse Engineering*, pages 154–163, Los Alamitos, 1998. IEEE Computer Society.

[30] R. Koschke, J.-F. Girard, and M. Würthner. An Intermediate Representation for Integrating Reverse Engineering Analyses. In *Fifth Working Conference on Reverse Engineering*, pages 241–250. IEEE Computer Society, Los Alamitos, 1998.

[31] R. L. Krikhaar. Reverse architecting approach for complex systems. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 4–11. IEEE Computer Society, 1997.

[32] J. Krinke and G. Snelting. Validation of Measurement Software as an Application of Slicing and Constraint Solving. *Information and Software Technology*, 40(12), 1998.

[33] B. Kullbach and A. Winter. Querying as an Enabling Technology in Software Reengineering. In P. Nesi and C. Verhoef, editors, *Proceedings of the 3rd Euromicro Conference on Software Maintenance & Reengineering*, pages 42–50, Los Alamitos, 1999. IEEE Computer Society.

[34] B. Kullbach, A. Winter, P. Dahm, and J. Ebert. Program Comprehension in Multi-Language Systems. In *Fifth Working Conference on Reverse Engineering*, pages 135–143. IEEE Computer Society, Los Alamitos, 1998.

[35] S. Liu and N. Wilde. Identifying objects in a conventional procedural language: an example of data design recovery. In *Proceedings of the International Conference on Software Maintenance 1990*, pages 266–271. IEEE Computer Society Press, 1990.

[36] A. Mendelzon and J. Sametinger. Reverse Engineering by visualizing and querying. *Software—Concepts and Tools*, 160(4):170–182, 1995.

[37] L. Moonen. A Generic Architecture for Data Flow Analysis to Support Reverse Engineering. In A. Sellink, editor, *Proc. 2nd Int. Workshop on the Theory and Practice of Algebraic Specifications*, Amsterdam, 1997. Springer-Verlag.

[38] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl. A Reverse Engineering Approach To Subsystem Structure Identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, Dec. 1993.

[39] S. Paul and A. Prakash. Querying source code using an algebraic query language. In *Proceedings of the International Conference on Software Maintenance 1994*, pages 127–136. IEEE Computer Society Press, Sept. 1994.

[40] A. Postma and M. Stroucken. Applying Relation Partition Algebra for Reverse Architecting. In J. Ebert and F. Lehner, editors, *Proceedings Workshop Reengineering, Bad Honnef, 27.-28. May 1999*. University of Koblenz-Landau, Koblenz, 1999.

[41] S. Rugaber and L. Wills. Creating a Research Infrastructure for Reengineering. In *Proceedings Third Working Conference on Reverse Engineering*, pages 98–102. IEEE Computer Society Press, Los Alamitos, 1996.

[42] A. Schürr. PROGRESS: A VHL-Language Based on Graph Grammars. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Graph grammars and their application to computer science*, volume 532 of *LNCS*, pages 641–659. Springer, Berlin, 1991.

[43] R. W. Schwanke. An Intelligent Tool for Re-engineering Software Modularity. In *Proceedings of the 13th International Conference on Software Engineering*, pages 83–92, May 1991.

[44] H. M. Sneed and E. Nyáry. Extracting Object-Oriented Specification from Procedurally Oriented Programs. In L. Wills, P. Newcomb, and E. Chikofsky, editors, *Proceedings of the Second Working Conference on Reverse Engineering (WCRE). Toronto, Ontario, Canada, July, 14-16 1995*, pages 217–226, Los Alamitos, California, 1995. IEEE Computer Society Press.

[45] G. Snelting. Concept analysis — A new framework for program understanding. *ACM SIGPLAN Notices*, 33(7):1–10, July 1998.

[46] J. F. Sowa. *Conceptual Structures. Information, Processing in Mind and Machine*. The Systems Programming Series. Addison-Wesley, Reading, 1984.

[47] J. M. Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice Hall, Englewood Cliffs, 2 edition, 1992.

[48] B. Steensgaard. Points-to analysis in almost linear time. In *POPL'96*, pages 32–41. ACM Press, Jan. 1996.

[49] M.-A. Storey and H. A. Müller. Manipulation and Documenting Software Structures Using SHriMP Views. In *Proceedings of the International Conference on Software Maintenance 1995*, pages 275–285. IEEE Computer Society Press, 1995.

[50] F. Tip. A Survey of Program Slicing Techniques. *Journal of programming languages*, 3:121–189, 1995.

[51] M. van den Brand, P. Klint, and P. A. Olivier. Compilation and memory management for ASF+SDF. Technical Report SEN-R9906, CWI - Centrum voor Wiskunde en Informatica, Feb. 28, 1999.

[52] M. van den Brand, A. Sellink, and C. Verhoef. Control Flow Normalization for COBOL/CICS Legacy Systems. In P. Nesi and F. Lehner, editors, *Proceedings of the 2nd Euromicro Conference on Software Maintenance & Reengineering*, pages 11–19, Los Alamitos, 1998. IEEE Computer Society.

[53] M. van den Brand, A. Sellink, and C. Verhoef. Current Parsing Techniques in Software Renovation Considered Harmful. In U. De Carlini and P. K. Linos, editors, *6th International Workshop on Program Comprehension*, pages 108–117. IEEE Computer Society, Washington, June 1998.

[54] Extensible Markup Language (XML) 1.0. W3c recommendation, W3C XML Working Group, http://www.w3.org/TR/1998/REC/xml/19900210, February 1998.

[55] P. Wegner. Dimensions of object-oriented modeling. *IEEE Computer*, 25(10):12–20, Oct. 1992.

[56] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[57] T. A. Wiggerts. Using Clustering Algorithms in Legacy Systems Remodularization. In I. Baxter, A. Quilici, and C. Verhoef, editors, *Proceedings of the 4th Working Conference on Reverse Engineering*, pages 33–43, Los Alamitos, 1997. IEEE Computer Society Press.

[58] K. Wong. RIGI User's Manual, Version 5.4.4. http://www.rigi.csc.uvic.ca/rigi/rigiframe1.shtml?Download, June 1998.

[59] S. Woods, L. O'Brien, T. Lin, K. Gallager, and A. Quilici. An architecture for interoperable programm understanding tools. In *Proceedings of the IWPC '98 (The Sixth International Workshop on Program Comprehension), Ischia, Italy, June 24-26, 1998*, 1998.