A Meta-model for Representing Language-independent Primary Dependency Structures

Ioana Şora
Department of Computer and Software Engineering, Politehnica University of Timisoara, Timisoara, Romania

Keywords: Reverse Engineering, Meta-models, Structural Dependencies.

Abstract: Reverse engineering creates models of software systems, at a higher level of abstraction or in a form suitable to a particular analysis. This article presents a meta-model that provides a unitary way of describing primary dependency structures in software systems. It extracts and conceptualizes similarities between different programming languages which, moreover, belong to any of the object-oriented as well as the procedural programming paradigms. The proposed meta-model is validated by the implementation of different tools for model extraction from programs written in Java, C# (CIL) and ANSI C. The utility of the proposed meta-model is shown by supporting a number of different analysis applications such as architectural reconstruction, impact analysis, modularization analysis, refactoring decisions.

1 INTRODUCTION

Reverse engineering software systems is, as defined in the seminal article (Chikofsky and Cross, 1990), the process of analyzing the subject system in order to identify the components of the system and the relationships between them and to create representations of the system in another form or at a higher level of abstraction.

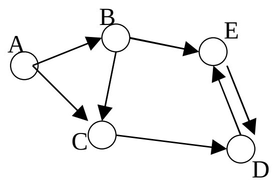
Reverse engineering typically comprises following steps: first, primary information is extracted from system artifacts (mainly implementation); second, higher abstractions are created using the primary information. The abstractisation process is guided by a particular purpose, such as: design recovery, program comprehension, quality assessment, or as a basis for reengineering. There is a lot of past, ongoing and future work in the field of reverse engineering (Canfora and Di Penta, 2007).

It is exactly because there exist so many contributions in the field of reverse engineering that it arises now a big new challenges in the field: to be able to integrate different tools and to be able to reuse existing analysis infrastructures in different contexts. The research roadmap of (Canfora and Di Penta, 2007) points out the necessity to increase tool maturity and interoperability as one of the future directions for reverse engineering. In order to achieve this, formats for data exchange should be unified and common schemas or meta-models for information representation must be adopted.

In this article, we propose a meta-model for representing language-independent and, moreover, programming paradigm independent, dependency structures. It arises from our experience of building ART (Architectural Reconstruction Toolsuite). We started by experimenting new approaches of architectural reconstruction of Java systems by combining clustering and partitioning (Sora et al., 2010). Soon we wanted to be able to use our reconstruction tools on systems implemented in different languages. Moreover, some of the languages addressed by the architectural reconstruction problem belong to the object-oriented paradigm, such as Java and C#, while other languages such as C are pure procedural languages, but this must not be a relevant detail from the point of view of architectural reconstruction. Later, the scope of ART extended to support also several different types of program analysis, such as dependency analysis, impact analysis, modularization analysis, structural comparison of projects for possible plagiarism detection. In order to support all of these goals, we have to extract and work with models that abstract the relevant traits of primary dependency relationships which occur in different languages, and are still able to serve different analysis purposes.

The remainder of this article is organized as follows. Section 2 presents background information about creating and using structural models in certain kinds of software reverse engineering applications.

ÈŸora I..
A Meta-model for Representing Language-independent Primary Dependency Structures.
DOI: 10.5220/0003991400650074
In *Proceedings of the 7th International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE-2012), pages 65-74
ISBN: 978-989-8565-13-6



	A	B	C	D	E
A					
B	3				
C	2	5			
D			1		4
E		1		6	

Figure 1: A dependency graph and a DSM.

Section 3 presents the proposed meta-model for representing primary dependency structures. Section 4 presents implementation and usage of the proposed meta-model. Section 5 discusses the proposed meta-model in the context of related work.

2 CREATING AND USING STRUCTURAL MODELS

In reverse engineering approaches, software structures are often modeled with help of graphs. A graph representing a software system consists of nodes modelling entities (program parts) and edges modelling relations between these. Such graph-based modelling techniques can be used at different abstraction levels, as identified in (Kraft et al., 2007): Low-level graph structures, representing information at the level of Abstract Syntax Trees; Middle-level graph structures, representing information such as such as call graphs and program dependence graphs; High-level graph structures, representing architecture descriptions.

The goal of our work in ART was to capture information relevant for describing the static structure of a system and the dependency relationships between static program entities. Control flow is not of interest in our approach, thus we are working at a middle-level of abstraction.

Figure 1 represents an abstract structural model of a software system. The representation of the model can be done as a directed graph or as a the corresponding Dependency Structure Matrix (DSM) (Sangal et al., 2005).

However, Figure 1 defines mainly a model representation syntax, and not a model semantics. Moreover, the same representation can have different semantics, according to the meaning associated with the nodes and edges.

If we assume that the nodes represent software components and the relationships model static dependencies between these, then such a DSM can serve as the basis for architecture reconstruction by identifying layers through partitioning (Sarkar et al., 2009), (Sangal et al., 2005) or subsystems through clustering (Mitchell and Mancoridis, 2006), (Sora et al., 2010). The program entities involved in this kind of architectural reconstruction are classes, when the system is implemented in the object-oriented paradigm (Java, C#) or modules (files) when applied to systems implemented in the structured programming paradigm (C). Relationships model in all cases code dependencies. However, in the object-oriented case, a dependency between two entities (classes) combines inheritance, method invocation, attribute accesses, while in the structured programming a dependency between two entities (modules) comes from accesses to global variables, function calls, use of defined types. Relationships can be characterized by their strength, leading to a weighted graph. Empirical methods may associate different importances to different kinds of relationships and define ways to quantify them.

We can also assume that the nodes in Figure 1 represent more fine-grained program entities, such as the members of a class, or elements of the same module. Relationships model facts such as a particular attribute being accessed by a particular method or a given method being called from another particular method. Based on this kind of model of the internal structure of a component, one can identify big components with low internal cohesion, and, by applying clustering algorithms, one can find refactoring solutions such as splitting. Clustering will lead to identify several smaller and more cohesive parts that the big and not cohesive component can be split into.

By putting together all these pieces, the extensible architecture of the ART tool-suite results as in Figure 2. The primary dependency structure models are the central element of it, and they introduce following major benefits:

- A primary dependency structure model abstracts code written in different programming languages and programming paradigms for which a Model Extractor Tool exists. All analysis tools are applied uniformly on extracted abstract models.
- A primary dependency structure model contains

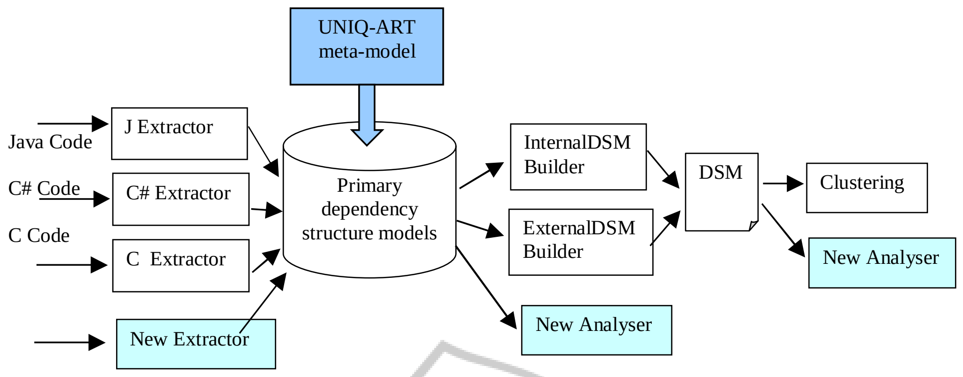


Figure 2: Architecture of the ART tool-suite.

enough semantical information of the problem domain (structure of software systems)such that it is able to serve different analysis purposes.

3 THE UNIQ-ART META-MODEL

3.1 Description of Our Meta-model

As concluded in the previous section, a reverse engineering toolsuite such as ART needs that primary dependency models are expressed according to an unique schema, independent of the programming language in which the modeled systems have been implemented. We define an unique meta-model for representing structural program dependencies, further called the UNIQ-ART meta-model.

Our general meta-modeling approach can be described as a 4-layered architecture, similar to the OMG's MOF (OMG, 2011a), as depicted in Figure 3. The next subsubsections present details of these layers.

3.1.1 The General Approach (the Meta-meta-Layer)

The meta-meta-level (Layer M3) contains the following concepts: *ProgramPart*, *AggregationRel*, *DependencyRel*. These are suitable for the analysis techniques of ART which need models able to represent different relationships between different program parts. The program parts refer to structural entities of programs, while the relationships are either aggregation relationships or dependency relationships.

Having the distinction between aggregation relationships and dependency relationships at the highest meta-level, this allows us to easily zoom-in and zoom-out the details of our models, (i.e., having a dependency model between classes it can be easily zoomed-out at package level).

3.1.2 The UNIQ-ART Meta-layer

The goal of this meta-layer (Layer M2) in UNIQ-ART is to identify similar constructs in different languages and even different constructs that can be mapped to the same meta-representation. Certain language particularities which are not relevant for dependency-based analyses are lost in this process, but it is a reasonable trade-off when taking into account the fact that it creates a large reuse potential for different analysis tools. The following paragraphs detail how we define these concepts and highlight some of the more relevant aspects of their mapping to concrete programming languages with examples related to Java and C.

ProgramPart Instances and Aggregation Relationships in the Meta-layer. The meta-meta-concept *ProgramPart* has following instances at the M2 level: *System*, *UpperUnit*, *Unit*, *ADT*, *Variable*, *Function*, *Parameter*, *LocalVar*, *Type*. These are the types of structural program parts defined in our meta-model. They are mapped to concrete particular concepts of different programming languages.

A *System* is defined as the subject of a reverse engineering task (a program, project, library, etc).

An *Unit* corresponds to a physical form of organization (a file).

An *UpperUnit* represents a means of grouping together several *Units* and/or other *UpperUnits*. It corresponds to organizing code into a directory structure (C) or into packages and subpackages (Java) or by name spaces (C#).

An *ADT* usually represents an Abstract Data Type, either a class in object-oriented languages or a module in procedural languages. Abstract classes and in-

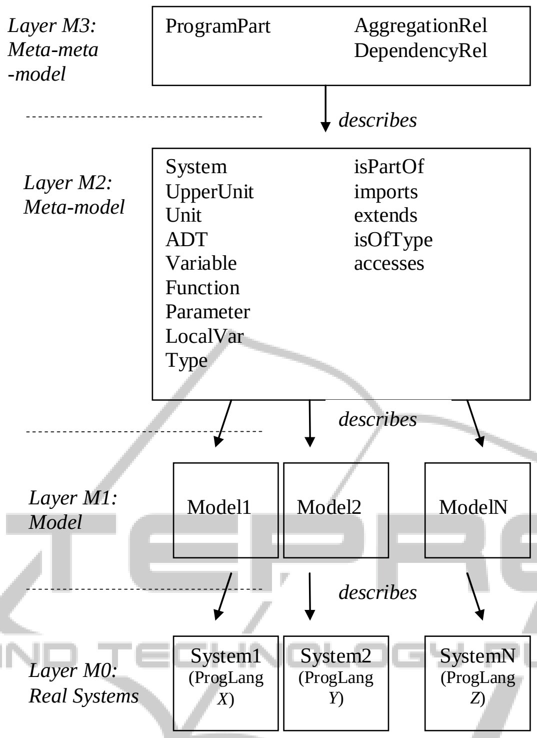


Figure 3: The UNIQ-ART meta architecture.

terfaces are also represented as *ADTs*. An ADT is always contained in a single unit of code, but it is possible that a unit of code contains different ADT's, as it is possible for C# code to declare several classes in a single file. In the case of purely procedural languages, when no other distinction is possible, the whole contents of a unit of code is mapped to a single *Default ADT*, as it is the case with C.

The relationships allowed between program parts at this level are aggregation relationships (*isPartOf*) and dependency relationships(*imports*, *extends*, *isOfType*, *calls*, *accesses*).

Between the different types of structural entities we consider following *isPartOf* aggregation relationships:

A *System* can be composed from several *UpperUnits*. Each *UpperUnit isPartOf* a single *System*.

An *UpperUnit* may contain several other *UpperUnits* and/or *Units*. Each *Unit isPartOf* a single *UperUnit*. There can be a *UpperUnit* which *isPartOf* a single another *UperUnit*.

A *Unit* may contain *ADTs*. Each *ADT isPartOf* a single *Unit*.

An *ADT* contains several *Variables*, *Functions*, *Types*. Each *Variable*, *Function* or *Type isPartOf* a single *ADT*.

An *ADT* corresponds to a module implementing an abstract data type, or to a class, or an interface. In case of procedural modules, the parts of the ADT (*Variables* and *Functions*) represent the global variables, functions and types defined here. In case of classes, the parts of the ADT (*Variables* and *Functions*) are mapped to the fields and methods of the class. Constructors are also represented by *Functions* in our meta-model.

A *Function* is identified by its name and signature. Representing instruction lists of functions is out of the scope of UNIQ-ART. The model records only whether functions declare and use local variables of a certain type. A *Function* contains several *Parameters* and/or *LocalVars*. *Parameter* objects represent both input and output parameters (returned types). Each *Parameter* or *LocalVar* object *isPartOf* a single *Function* object in a model.

Dependency Relationship Instances in the Meta-layer. Besides the aggregation relationships, program parts have also dependency relationships.

The dependency relationships considered here are: *imports*, *extends*, *isOfType*, *calls*, *accesses*.

Import relationships can be established at the level of a *Unit*, which may be in this relationship with any number of other *Units* or *UpperUnits*. It models the situations of using packages, namespaces or including header files.

An *ADT* may also *extend* other *ADTs*. This corresponds to the most general situation of multiple inheritance, where a class can extend several other classes. Extending an abstract class or implementing an interface in object oriented languages which allow this feature is also described by the extend relationship.

In the procedural paradigm, in a language such as C, we consider that the relationship established between a module and its own header file is very similar to implementing an interface and thus map it in our model in this category of *extends* relationships. In consequence, the fact of having a file include another file, will be generally mapped to an *import* relationship, but if the included file is its own header file (it contains declarations of elements defined in the importing file), the fact will be mapped to an *extend* relationship.

Each *Variable*, *Parameter* or *LocalVar* has a type either given by a *ADT*, a *Type* or a primitive construct built-in the language.

The interaction relationships *calls* and *accesses* and can be established between *Functions* and other *Functions* or between *Functions* and *Variables*.

The program parts of a system can establish relationships, as shown above, with other program parts of the same system. It is also possible that they establish relationships with program parts which do not belong to the system under analysis. For example, the called functions or accessed variables do not have to belong to units of the same system: it is possible that a call is made to a function which is outside the system under analysis, such as to an external library function. Another example is a class that is defined in the system under analysis but which extends another class that belongs to an external framework which is used. Such external program parts have to be included in the model of the system under analysis, because of the relationships established between them and some internal program parts. However, the models of such external program parts are incomplete, containing only information relevant for some interaction with internal parts. Units that are external to the system are explicitly marked as external in the model.

3.2 An Example

In this section we illustrate the layers M1 and M0 of UNIQ-ART (see Figure 3) on an example. For this presentation purposes, we assume that the modeled real system, `LineDrawings`, is implemented in a language presenting both object-oriented and procedural characteristics - permitting both the definition of classes as well as the use of global variables and functions. In order to present as many features of UNIQ-ART in a single example, we combine features of Java and C in the hypothetical language used in the example system `LineDrawings` shown in Figure 4.

The model of the LineDrawings system is depicted in Figure 5.

The code of the example system is organized in two folders, `Figures` and `Program`, represented in the model as *UpperUnits*. These folders contain the source code files `SimpleFigures` and `Drawing`, which are represented by the two *Units*.

The `SimpleFigures` unit contains two classes, `Point` and `Line`, represented as *ADTs* belonging to the `SimpleFigures` *Unit*.

The class `Point` has two fields, X and Y, represented as *Variable* objects of the meta-model. Both are of a primitive type and thus not further captured by our model. The class `Point` has a constructor `init()` and a member function `show()`, both of them represented as *Function* objects of the meta-model. The constructor accesses both fields, represented by the *access* relationships. The `init()` function takes two parameters, which are of a primitive type and thus they introduce no *isOfType* relationships in our model.

The class `Line` has two fields, P1 and P2, represented in the model as Variables that are part of the `Line` *ADT*. These variables are of the type `Point` that has been already represented as the *ADT* `Point` as described above. This fact is reflected in the *isOfType* relationships. Class `Line` has a constructor `init()` and member functions `draw()` and `nextPoint()`. All functions access both fields, shown in the *access* relationships between them. The constructor of `Line` *calls* the contructor of `Point`. The function `draw()` has a local variable P which is of type `Point`, this is represented in the model by a *LocalVar* which *isPartOf* the *Function* `draw`. The *Function* `draw` *calls* functions `show()` defined in *ADT* `Point` and `nextPoint` defined in *ADT* `Line`.

The `Drawing` file defines no classes. According to the UNIQ-ART modeling conventions, in the model, the `Drawing` *Unit* contains a single `default` *ADT* which contains only one global function, `main()`. The function `main()` *calls* the constructor of *ADT* `Line` and *calls* their `draw()` function.

The Function `main()` contains two *LocalVar* parts, each of them associated with the *ADT* `Line` through an *isOfType* relationship.

The function `main()` of the `Drawing` Unit also *calls* function `fgraph()` from an external library `Initgraph` (which is not part of the system under analysis, which is the system LineDraw). The Unit `Initgraph` has been not included in the code to be modeled thus it is considered external to the project

File SimpleFig.jc

```
package Figures;

class Point {
      public int X, Y;
      public Point(int A, int B) {
            X = A; Y = B;
      }
      public void show() {
      // implementation omitted
      }
}

public class Line {
      public Point P1, P2;
      public Line(int A, int B, int C, int D) {
            P1 = new Point(A, B);
            P2 = new Point(C, D);
      }
      public void draw() {
            Point P;
            for (P = P1; P != P2; P = nextPoint(P))
                  P.show();
      }
      private Point nextPoint(Point P) {
            return new Point(P.X+1, P.Y+(P2.X-P1.X)/(P2.Y-P1.Y));
      }
}
```

File Drawing.jc

```
#includes "initgraph.h"
import Figures.SimpleFig;

void main(void) {
      fgraph();
      Line L1 = new Line(2, 4, 7, 9);
      Line L2 = new Line(5, 8, 12, 18);
      L1.draw();
      L2.draw();
      }
```

Figure 4: Example of code (hypothetical Java-C).

under analysis, having only a partial model. We know only one function, `fgraph()`, out of all the elements belonging to this Unit, and only because it is called by a function that is part of our system.

This choosen example covered many of the features of UNIQ-ART. One important feature of UNIQ-ART which has not been covered by this example is the *extends* relationship which is used to model class inheritance, interface implementation or own header file inclusion. Also, another feature of UNIQ-ART not illustrated by this example is modeling simple type definitions which are not ADTs (`typedef structs` in C for example).

4 IMPLEMENTING AND USING UNIQ-ART

4.1 Representing the UNIQ-ART Meta-model

Models and their meta-models can be represented and stored in different ways: relational databases and their database schema, XML files and their schema, logical fact bases and their predicates, graphs.

For UNIQ-ART, we have choosen a relational database (MySQL) with an adequate schema of tables and relationships. Such a platform allows us to integrate model extractor tools from different languages and platforms. However, if interaction with other tools would require it so, a UNIQ-ART model can be easily mapped to other means of representa-

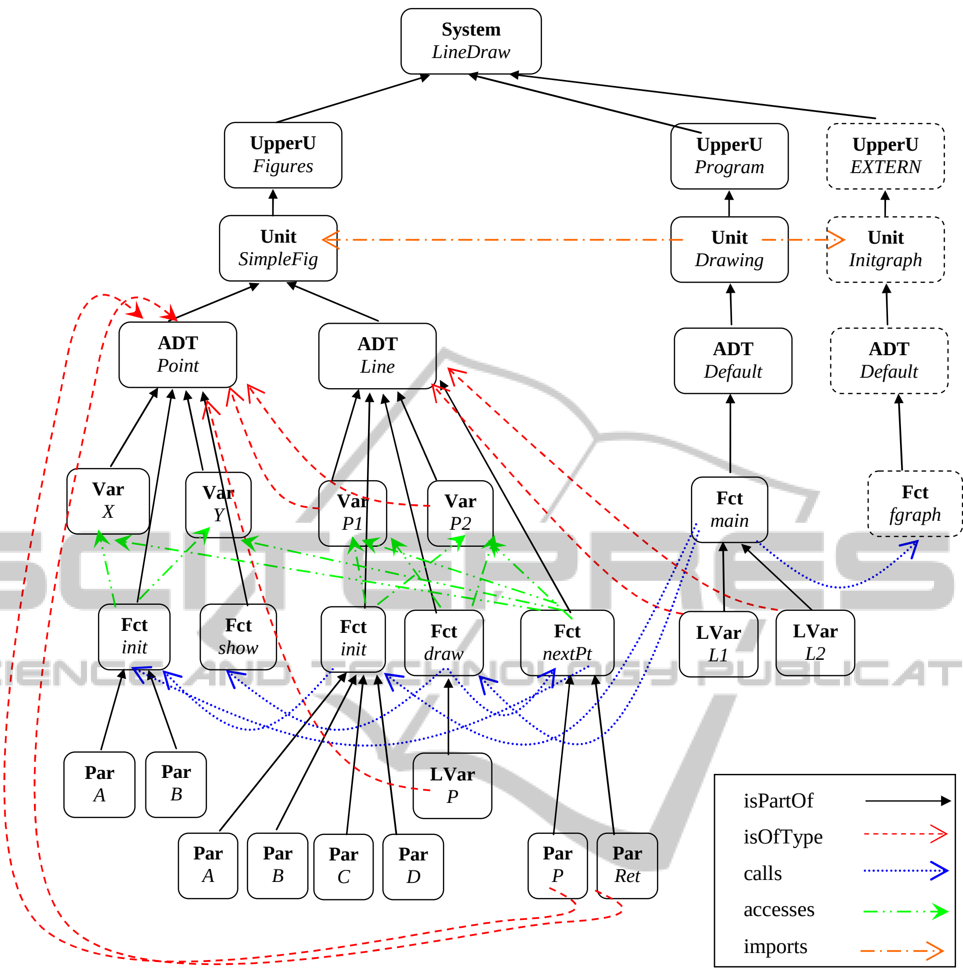


Figure 5: Example of a model.

tion without loosing generality, for example a GXL (Kraft et al., 2007) description could be easily generated starting from the contents of a database or directly by adapted model extractor tools.

Tables corresponds to each type of *ProgramParts* (*Systems*, *UpperUnits*, *Units*, *ADTs*, *Functions*, *Parameters*, *LocalVars*, *Variables*, *TypeDefs*). The *isPartOf* relationship, which is defined for each structural element, is represented by foreign key aggregation (n:1 relationship). The *isOfType* relationship is defined only for *Variables*, *LocalVars* and *Parameters*. In these tables, there is first a discriminant between the cases whether it is a primitive type, an *ADT*, or a *Typedef* and, in the case of not primitives, there is a foreign key association with a row in the corresponding table, ADTs or TypeDefs (n:1). The other relationships - *extends*, *calls*, *accesses* - are of cardinality n:m and are represented by association tables.

4.2 Model-extractor Tools

Model extractor tools for Java, C# and ANSI C are implemented in the ART toolsuite. Each model extractor is a completely independent tool, implemented using its own speciffic language, framework or technologies. Moreover, two of our model extractor tools work on compiled code (Java bytecode and Microsoft CIL) and can extract all the information they need.

The Java model extractor tool works on bytecode, processed with help of the ASM Java bytecode analysis framework (http://asm.ow2.org/).

Another model extractor tool works on managed .NET code, being able to handle code coming from any of the .NET programming languages (i.e. C#, VB, etc.) that are compiled into managed code (compiled into CIL - Common Intermediate Language). CIL is an object-oriented assembly language, and its object-oriented concepts are mapped onto the concepts of our meta-model. The model extractor tool uses Lutz Roeders Reflector for .NET for inspecting the CIL code (http://www.lutzroeder.com/dotnet/).

The model extractor tool for C works on source code. Our implementation runs first srcML (Maletic et al., 2002) as a preprocessor in order to obtain the source code represented as a XML file which can be easier parsed. The procedural concepts of C are mapped into concepts of our meta-model. We recall here only the more specific issues: Files are the Units of the model as well as the ADTs (each unit contains by default one logical unit); The relationship between two units, one that contains declarations of elements which are defined in the other unit, is equivalent with extending an abstract class.

All model extractor tools populate the tables of the same MySQL database with data entries. For each system that will be analyzed, the primary model is extracted only once, all the analysis tasks are performed by starting from the model stored in the database.

The model extractor tools have been used on a large variety of systems, implemented in Java, C# and C, ranging from small applications developed as university projects until popular applications available as open-source or in compiled form, until, as the benchmark for the scalability of the proposed approach, the entire Java runtime (rt.jar). The execution times needed for the extraction of the primary model and its storage in the database are, for an average-sized system of cca 1000 classes, in the range of one minute. Taking into account that ART is a toolsuite dedicated to off-line analysis, like architectural reconstruction, these times are very reasonable once-for-a-system times. The extraction of the model and its storage in the database for a very large system (the Java runtime, having 20000 classes) took 24 minutes.

4.3 Applications using UNIQ-ART

A primary dependency structure model stored in the database could be used directly by writing SQL queries. For example, such queries could retrieve the *ADT* containing the most functions, or retrieve the *Function* which is called by most other functions, etc.

Another way of using a primary model is with help of dedicated analysis tools, or deriving more specialized secondary models and using specialized tools to further analyze these. This corresponds to the scenario that is most characteristic to the ART toolsuite as it has been depicted in Figure 2.

Some of the most frequent used secondary models in ART are different types of DSMs (dependency structure matrixs). The primary model supports building different types of specialized DSM's by choosing which of the program elements are exposed as elements of the DSM. Each kind of DSM generates a different view of the system and can have different uses. The two most frequent used DSM's in ART are the *external DSM*(between logical units), and the *internal DSM* (between elements belonging to the same logical unit).

In case of the *external DSM*, the *ADTs* of the primary model are exposed as the elements(rows and columns) of the DSM. The DSM records as the relationship between its elements at column i and row j the composition of all interactions of elements related to ADT_i and ADT_j. In this composition of interactions can be summed up all or some of the following: Variables contained in ADT_i which are of a type defined in ADT_j, Functions contained in ADT_i that contain LocalVars or Parameters of a type defined in ADT_j, the number of Functions contained in ADT_j which are called by Functions contained in ADT_j, the number of functions which call functions from both of ADT_i and ADT_j, the fact that ADT_i extends ADT_j, etc. Also, this relationship can be quantified, the strength of the relationship results by applying a set of different (empirical) weights when composing interactions of the aforementioned kinds. Such a DSM is further analyzed by clustering for architectural reconstruction, by partitioning for identifying architectural layers, for detecting cycles among subsystems, for impact analysis, modularity analysis, etc. Some of our results were described in (Sora et al., 2010).

In case of the *internal DSM* of a *ADT*, the program parts contained in the ADT become the elements (rows and columns) of the DSM. The rows and columns of the DSM correspond to *Variables* and *Functions* of a ADT. The existence of a relationship between the elements at column$i$ and row j is given by composing different possible interactions: direct call or acces relationships between elements i and j, the number of other functions that access or call both of the elements i and j, the number of other variables accessed by both of the elements i and j, the number of other functions called by both of the elements i and j, etc. Such a DSM is used for analyzing the cohesion of an unit, and it can be used for taking refactoring decisions of splitting large and uncohesive units.

The times needed to build a secondary model (a DSM) in memory, starting from the primary model data stored in the database, are approximatively 10 times smaller than the time needed initially for extracting the primary model.

5 RELATED WORK

There is a lot of previous and ongoing work in the fields of reverse engineering addressed also by ART, dealing with subjects like modularity checking (Wong et al., 2011), layering (Sarkar et al., 2009), detection of cyclic dependencies (Falleri et al., 2011), impact analysis (Wong and Cai, 2009), clustering (Mitchell and Mancoridis, 2006). They all have in common the fact that they build and use some dependency models which are conceptually similar with Dependency Structure Matrixes (Sangal et al., 2005). Our work in building ART (Architectural Reconstruction Toolsuite) (Sora et al., 2010) is also in this domain.

One problem that has been noticed early in the field is that existing tools and approaches are difficult to reuse in another context and that existing tools are difficult to integrate in order to form tool-suites. In order to achieve this, two aspects have to be covered: first, formats for data exchange should be unified, and second, common schemas or meta-models for information representation must be adopted.

Tools can be adapted to use a common data format. In (Kraft et al., 2007), an infrastructure to support interoperability between tools of reverse engineering assumes that software reengineering tools are graph based tools that can be composed if they use a common graph interchange format, GXL (the Graph eXchange Language). GXL was developed as a general format for describing graph structures (Winter et al., 2002). But GXL does not prescribe a schema for software data. GXL provides a common syntax for exchanges and features for users to specify their own schema.

For example, in the case of the typical ART scenario depicted in Figure 2, the clustering tool can require that DSM is represented in GXL format. This will make the clustering tool more reusable on different data. However, the GXL syntax does not capture anything about the semantics of the DSM, whether it is an external DSM or an internal DSM, as it was discussed in Section 4.3.

Establishing only the common exchange format does not offer the needed support for extracting models of a similar semantics out of different kinds of system implementations. In order to fully achieve this, common schemas or meta-models for information representation must be used. It is with regard to this aspect that we introduce UNIQ-ART.

Reference schemas for certain standard applications in reverse enginneering have been developed. For example, such meta-models for C/C++ at the low detail (abstract syntax tree) level are proposed in (Ferenc et al., 2002) (the Columbus schema).

Some more general schemas for language-independent modelling, address the family of object oriented systems. Examples include the UML meta-model (OMG, 2011b) and the FAMIX meta-model (Tichelaar et al., 2000). They present similarities between them, as UML can be considered a standard for object-oriented concepts.

Compared by complexity of the meta-model and level of details that it is able to capture, UNIQ-ART is most similar with FAMIX (Tichelaar et al., 2000). FAMIX is used by a wide variety of re-engineering tools comprising software metrics evaluation and software visualization. FAMIX provides for a language independent representation of object-oriented source code, thus its main concepts are *Class*, *Method*, *Attribute*. FAMIX represent relationships between these also as entities *InheritanceDefinition*, *Access* and *Invocation*. It does not treat in any way procedural aspects (global variables and functions, user-defined types which are no classes) although probably it could be extended to do so. We defined the main concepts of UNIQ-ART and their language mappings in such a way that they *all* apply *transparently* to both object-oriented and procedural language concepts, since it was a main goal of the ART project to develop architectural reconstruction tools applicable for both object oriented and procedural systems.

Another metamodel is the Dagstuhl Middle Metamodel DMM (Lethbridge et al., 2004). It can represent models of programs written in most common object-oriented and procedural languages. But DMM provides the modelling capabilities for object-oriented and non-object-oriented modelling in an *explicit* way. DMM generalizes several concepts to achieve multi-language transparency, but not programming paradigm transparency. For example, in DMM a *Method* is a concept which is different from *Routine* or *Function*, although they are very similar, except for the fact that a *Method* has a relationship to a *Class*. This leads to an increased complexity of the DMM model, which contains a big number of *ModelObject* types and *Relationship* types. In contrast, our model started from the key decision to abstract away as many differences between the object-oriented and the procedural programming paradigm. As described in Section 3, the UNIQ-ART meta-model contains only nine different types of *ProgramParts* and six different types of relationships, all of them applying transparently to both object-oriented and procedural concepts. Since it operates with a small number of program part types, UNIQ-ART is lightweight and thus easy to use; however, it can be used by a number of different kinds of applications (architectural reconstruction of subsystems, identification of architectural layers, detection of cyclic dependencies, impact analysis, modularity analysis, refactoring of uncohesive modules by splitting, etc), which proves its modelling power and utility.

6 CONCLUSIONS

This article proposes UNIQ-ART, a meta-model that can represent primary dependency structures of programs written in object-oriented as well as procedural languages. UNIQ-ART achieves not only programming language transparency but also programming paradigm transparency. All model entities introduced by UNIQ-ART abstract away differences between object oriented and procedural concepts. We have implemented language mappings and model extractor tools for Java, C# (MS CIL) and ANSI C. The utility of the proposed meta-model has been shown by a number of different reverse engineering and analysis applications that use it successfully.

ACKNOWLEDGEMENTS

Acknowledgements go to the students which contributed to the implementation of the model extractor tools for different languages, especially Ramona Croitoru and Natalia Prica.

REFERENCES

Canfora, G. and Di Penta, M. (2007). New frontiers of reverse engineering. In *Future of Software Engineering, 2007. FOSE '07*, pages 326 –341.

Chikofsky, E. and Cross, J.H., I. (1990). Reverse engineering and design recovery: a taxonomy. *Software, IEEE*, 7(1):13 –17.

Falleri, J.-R., Denier, S., Laval, J., Vismara, P., and Ducasse, S. (2011). Efficient retrieval and ranking of undesired package cycles in large software systems. In *Proceedings of the 49th international conference on Objects, models, components, patterns*, TOOLS'11, pages 260–275, Berlin, Heidelberg. Springer-Verlag.

Ferenc, R., Beszedes, A., Tarkiainen, M., and Gyimothy, T. (2002). Columbus - reverse engineering tool and schema for C++. In *Software Maintenance, 2002. Proceedings. International Conference on*, pages 172 – 181.

Kraft, N. A., Malloy, B. A., and Power, J. F. (2007). An infrastructure to support interoperability in reverse engineering. *Information and Software Technology*, 49(3):292 – 307. 12th Working Conference on Reverse Engineering.

Lethbridge, T. C., Tichelaar, S., and Ploedereder, E. (2004). The Dagstuhl Middle Metamodel: A Schema For Reverse Engineering. *Electronic Notes in Theoretical Computer Science*, 94(0):7 – 18. Proceedings of the International Workshop on Meta-Models and Schemas for Reverse Engineering (ateM 2003).

Maletic, J., Collard, M., and Marcus, A. (2002). Source code files as structured documents. In *Program Comprehension, 2002. Proceedings. 10th International Workshop on*, pages 289 – 292.

Mitchell, B. S. and Mancoridis, S. (2006). On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32:193–208.

OMG (2011a). The metaobject facility specification. http://www.omg.org/mof/.

OMG (2011b). The Unified Modelling Language. http://www.uml.org/.

Sangal, N., Jordan, E., Sinha, V., and Jackson, D. (2005). Using dependency models to manage complex software architecture. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 167–176, New York, NY, USA. ACM.

Sarkar, S., Maskeri, G., and Ramachandran, S. (2009). Discovery of architectural layers and measurement of layering violations in source code. *J. Syst. Softw.*, 82:1891–1905.

Sora, I., Glodean, G., and Gligor, M. (2010). Software architecture reconstruction: An approach based on combining graph clustering and partitioning. In *Computational Cybernetics and Technical Informatics (ICCC-CONTI), 2010 International Joint Conference on*, pages 259 –264.

Tichelaar, S., Ducasse, S., Demeyer, S., and Nierstrasz, O. (2000). A meta-model for language-independent refactoring. In *Principles of Software Evolution, 2000. Proceedings. International Symposium on*, pages 154 –164.

Winter, A., Kullbach, B., and Riediger, V. (2002). An overview of the GXL graph exchange language. In Diehl, S., editor, *Software Visualization*, volume 2269 of *Lecture Notes in Computer Science*, pages 528–532. Springer Berlin / Heidelberg.

Wong, S. and Cai, Y. (2009). Predicting change impact from logical models. *Software Maintenance, IEEE International Conference on*, 0:467–470.

Wong, S., Cai, Y., Kim, M., and Dalton, M. (2011). Detecting software modularity violations. In *Proceeding of the 33rd international conference on Software engineering*, ICSE '11, pages 411–420, New York, NY, USA. ACM.