Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 94 (2004) 39–49
www.elsevier.com/locate/entcs

A Programmable Analysis and Transformation Framework for Reverse Engineering

Ravindra Naik[1] and Arun Bahulkar[2]

Tata Research Development and Design Centre
54, Industrial Estate, Hadapsar
Pune, INDIA

Abstract

Reverse Engineering refers to understanding an existing software system, and documenting the understanding in a desired format. Depending upon the purpose, which can be problem diagnosis, impact analysis, functional enhancements, or re-engineering to newer technologies or newer design, the reverse engineering artifacts vary in their definition, the abstraction level, and contents. Current state of practice relies on pre-cast tools, which extract a set of pre-defined information, with little or no inputs from the analyst. We propose a specification-based approach that allows analyst to specify and control the contents and abstraction level of the reverse engineering artifacts, while allowing user-interaction during extraction.

Keywords: Programmable Reverse Engineering, Reengineering, Program Analysis, Transformations

1 Reverse Engineering

Reverse engineering involves understanding an existing software system, and documenting the understanding in the desired format. Reverse engineering of a system may be undertaken for different purposes, viz.

- Understanding different design views of the system
  - To enable re-development of the system using newer technologies

[1] Email: rdnaik@pune.tcs.co.in
[2] Email: arun@pune.tcs.co.in

doi:10.1016/j.entcs.2004.01.005

  - To re-organize and re-structure the system so as to reduce the maintenance costs and prolong the life of the system
- Understanding the system for impact analysis / problem diagnosis in case of ongoing maintenance of an existing system
- Understanding the system for the purpose of extracting test specifications for the system

This activity involves extraction of artifacts at program-level like cross-references, data slice, program slice, program chops, etc. System level artifacts can be call-graphs, entity-relationship model, data-flow models, CRUD matrices, etc. Restructuring requirements need to locate code that implements different aspects like user-interface, business and reporting logic, database access, etc. Automating extraction of these artifacts is desirable. More desirable is the ability for tool builders to be able to specify the extraction of the artifacts, how to view the artifacts, and how to interact and guide the extraction process.

2 State of practice

Most existing reverse engineering tools provide either predefined artifacts as output, or provide raw analysis information of programs that can be used by analysts for reverse engineering. Examples are [1] and [2].

Few frameworks exist, that are based on the extract-abstract-view principle. Many allow querying over program sources and their structural properties, but do not support querying of semantic properties. Some of these frameworks are exploratory, while others are built for specific modern languages like C, C++ and Java. Examples are Acacia [3], CppAnal [4], and DALI [5].

Our approach differs primarily in that it provides mechanisms to define abstractions, to query on semantic program properties, and enable displaying the results in different forms appropriate for the purpose. The subject programs are represented as models. This provides an inherent, first level of abstraction, and makes the framework to be language independent. In addition, the framework is based on a proven, flow-analysis framework for procedural languages, Darpan [6]. Though the origin of this approach lies in our experience of building re-structuring and reverse engineering tools for COBOL and RPG applications, few lessons are derived from the work of Programmable Reverse Engineering [7]. Our work comes closest to the work by Stan Jarzabek and Guosheng Wang [10]. However, we have imperative style of specifications as against their declarative style, we support program semantic properties and aim to provide code level transformations.

3 Our approach

Programmable Analysis and Transformation Framework is a meta-framework, being developed for exploring and implementing reverse engineering tools that will enable program and system understanding, and their re-structuring.

The basic approach of our framework is to enable definition of abstractions (what-part), specify the way to extract them (how-part), and the way to view them (display-part). Codegen specifications will use the abstraction results to produce restructured programs from the source code.

The definition of abstraction involves specifying the composite type of the abstraction. This creates a representation for the abstraction, which is in terms of the model elements that represent a program.

The "how part" involves search over semantic space of program properties, and includes:

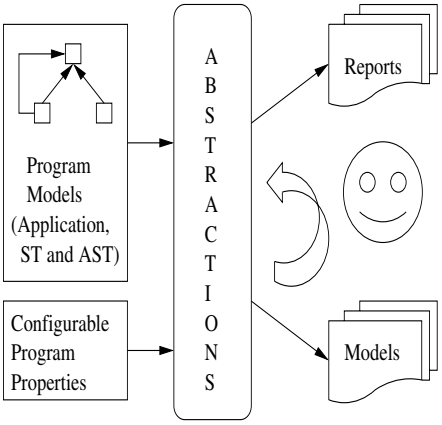
- syntax patterns based on syntactic pattern matching of the subject programs. These are realized by search over the model that represents the subject program.
- Extraction specifications can be seen as consisting of two parts:
  - Specifications that enable search of semantic patterns: these are queries of structural and static semantic properties of the source programs, and use of these properties to identify the abstraction of interest
  - Specifications to generate the elements that represent the desired abstraction

The codegen specifications will enable conversion of subject programs, based on the abstractions.

Display specifications enable mapping between abstraction definition and pre-defined display types. These enable the abstractions to be viewed using the specified displays. Alternately, user can view the abstractions in a manually programmed user-interface, or add new display types to the framework.

Though the current focus is on analysis of Mainframe technologies (JCL, COBOL, DB2), the framework is equally capable to support other programming technologies, like CICS, IMS, PL/I, C, etc.

4 Framework meta-model



We view the source programs as models, thereby raising the level of abstraction of the input. Core program analysis is viewed as a process of computing pre-defined program properties from program models. Program properties, like control-flow graph, data-flow information can also be viewed as abstractions.

Armed with these program properties, our framework allows search over source code (read program models), query of the program properties and generation of abstractions (whose elements are program model objects or compositions thereof), predicated by the program properties.

The program models are generic program representations, which are suitable for programs written in procedural or OO programming languages. The models have placeholders for types and definition of symbols represented by symbol table [6]. The operations and computations are realized through various constructs of the programming language and are represented by abstract syntax tree [6].

The symbols and operations for database constructs, specific technologies on Mainframe machines (Job Control Language, CICS transaction manager, etc.) are represented using extensions of the symbol table and abstract syntax tree. Finally, an application is a composition of all symbol tables and abstract syntax trees.

Search over the source code is supported by providing a bottom-up and top-down search of the program model. The search yields program elements (like variables, functions, declarations) of interest.

The framework makes program properties available as primitives. Program properties like call-graph, def-use chains, alias information, copy constants, and others are computed by a data-flow analyzer, which is generated using Darpan [6]. New properties can be computed by writing new specifications for Darpan or can be provided by the user. The idea is to apply these properties to the program elements of interest, and based on the property values, decide whether the program element can belong to the abstraction being constructed.

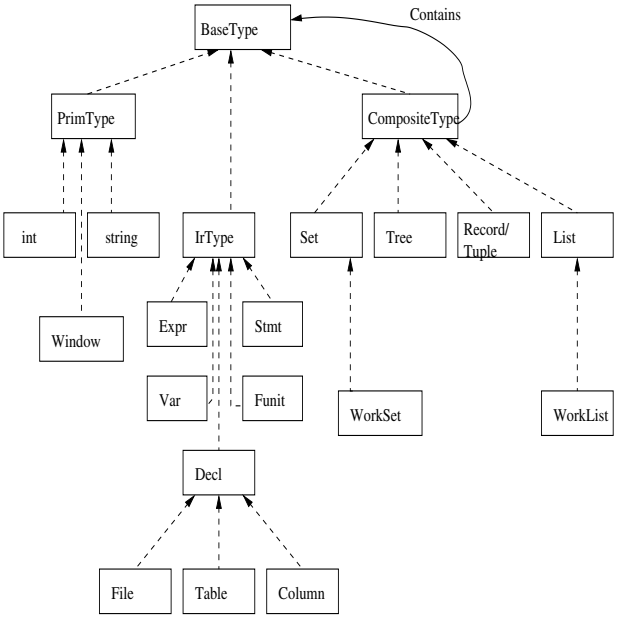


Figure. 1

For example, in COBOL, while constructing ER model, if two logical files are always mapped to the same physical file, then the combination represents a single entity, else it represents to two independent entities. The type system for defining abstractions and accessing the program properties is shown in Figure 1.

Abstractions so extracted should be displayed in a user-defined view. For this, our framework allows abstraction types to be mapped to display types. Currently supported display types are shown in Figure 2. The display types are editable (user can edit the displayed results), which enables the abstraction process to be interactive and iterative.

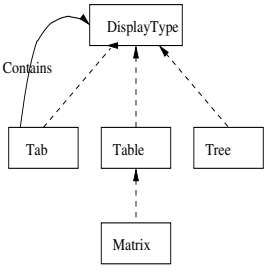


Figure. 2

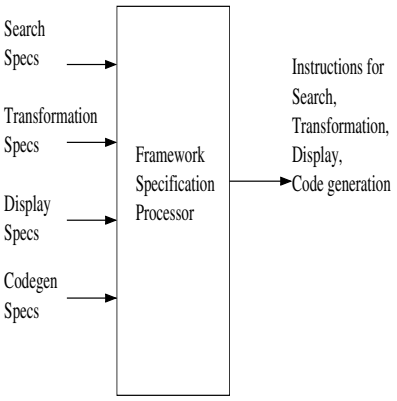
Summarizing, the abstraction process is, in general, iterative over the program models and program properties. User terminates the iterations when he is satisfied with the abstraction contents. This process can be viewed as a model transformer, which accepts the model representing the source programs as input, and allows analysts to specify transformations to generate models representing program understanding.

5 Framework Architecture

The framework consists of two logical activities:

- Specification Processing and Validation
- Specification Execution

5.1 Specification Processing and Validation



Specification processing checks syntax of the specifications, and types of the symbols and expressions used in the specifications. This step generates JVM instructions that can be executed by Java Virtual Machine.

We are in the process of conceptualizing and defining the specifications for generating models, and the codegen specifications. We plan to base these transformation specifications using MOF-QVT standards [9].

(i) Search pattern specifications

These are patterns that allow search over source code of programs. Source code elements like declaration, variable, function, statement, data files, database tables, and others can be specified as search elements.

For example, to extract error-handling code, a search pattern is written that fetches all declarations of variables that have the string error in their name. This is written as:

```
select $decl where ( namelike "error" ) $PROAXROOT
```

(ii) Transformation specifications

These are specifications that allow logical search over semantic properties of programs. They allow queries of flow analysis (control and data) properties of the subject programs, and allow various kinds of traversals and compositions of these properties. They support primitive types like integer, string, Boolean; model types to represent source code elements like statement, expression, declaration, variable, function; composite types like sets, lists, records and operations like union, intersection, enumeration. Imperative constructs for sequencing, conditional, and iteration, are supported. Modularization of specifications is supported through function construct.

For example, to extract error-handling code, one activity is to extract the variable references in the subject program, which occur in l-value context. This has to be done for each variable declaration that is a prospective error message variable. This is written as:

```
  for each adecl in declSet
{
    rvarSet = refers( adecl, $PROAXROOT );
```

```
    for each rvar in rvarSet
    {
        if ( isComputed( rvar ) )
            lvalVarSet += rvar
    }
}
```

Two analysis primitives, viz. refers (returns all variable references of a declaration passed as 1st parameter in the scope identified by 2nd parameter), and isComputed (returns true if the variable reference passed as parameter is used in l-value context) are used in the example.

(iii) Display specifications

These specifications allow users to specify two kinds of information. One is how the user would see the results of transformation specifications; this part consists of window specifications. Second is a mapping specification to map the abstraction types to pre-defined display types. They are specified as XML specifications. An example mapping specification to display error-handling statements is written as:

```
<Map id = "eeh_res_tree">
    <Abstraction-Type>
        <Set>
            <Stmt  id= "n1" />
        </Set>
    </Abstraction-Type>
    <Display-Type>
        <DTree connected="false">
            <Root-Node Heading="Error handling statements">
                <Tree-Node>
                    <Tree-Node Data="n1" />
                </Tree-Node>
            </Root-Node>
        </DTree>
    </Display-Type>
</Map>
```

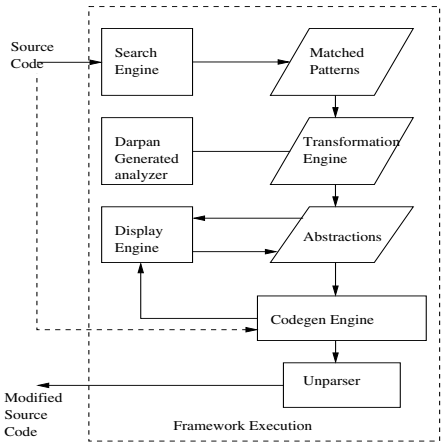
This specification maps the statement type to Tree type, and shows all error handling statements as nodes of a tree. Tables and Matrix are other display types supported.

The second kind of display specification is an operational specification of tool execution. This specification is an XML specification, and allows the execution environment to set up menus in the GUI, which (when clicked) initiate execution of the specified tool.

(iv) Codegen specifications

These specifications will allow analysts to specify transformation of the source programs to desired target programs. This will be based on analysis abstractions extracted earlier. One example is to separate the source programs into GUI layer and business layer. Code generation capability will need to generate concrete syntax of the target programs.At the moment, we are working on these specifications.

5.2 Specification Execution



The analysis and transformation framework is designed to be interpretive. The sole purpose is that the analysts can build reverse engineering tools quickly, and explore by changing the tool specifications and viewing the changed behaviour of the tool. Various engines are provided to support the execution environment: search engine, abstraction engine, display engine and data-flow analyzer.

The framework can be viewed as a model transformer, which accepts the model representing the source programs as input, and allows analysts to specify transformations to generate models representing program understanding.

6 Status

Having reused existing parsers, model representation of programs and analyzer generator, the framework is currently in a prototype stage. Using the prototype, we are defining ER (Entity-Relationship) model extraction from COBOL programs. We propose to use this framework for DFD (Data-Flow) model extraction, both of these geared towards system documentation initiative.

We plan to extend the framework by supporting model types, and code generation. We also plan to add more viewing primitives. We hope that we can use third-party softwares, especially for viewing, by supporting standard exchange formats like [8].

7 Implementing a reverse engineering tool using the framework

We describe below the steps involved in defining and implementing a reverse engineering tool using the framework. The steps mentioned here are indicative.

The analyst identifies the abstractions that are to be delivered (e.g. Call graph report, Error handling statements, Entity-Relations model, etc.). For each abstraction, analyst defines the representation / type.

Analyst creates the GUI specifications for windows to be used, menus, menu-items and icons for the same, and associate the menu-items / icons with the name of tool specification (to be written next).

Analyst writes the search and transformation specifications for the tools. Simultaneously, he writes the mapping specifications for the display and required user interaction.

Analyst implements the non-framework-based-tools (if any). This implementation needs to comply with the type-system of the framework and interfaces.

Analyst starts execution of the framework, which has a pre-defined user-interface. In case of framework-based-tool, analyst builds the tool. This runs the specification processor, generates interpreter instructions, and adds the tool to the menus of the framework. In case of non-framework-based-tool, analyst uses the build menu to specify mapping between tool and the class that implements the tool. The build will add the tool to the menus of the framework.

8 Sample specification

Included is a sample specification for extraction of Entities, Attributes for data files of COBOL. This specification treats each logical file that is accessed in a COBOL program as an entity, and its (record's) layout as attribute. Other necessary functions are not included in this example, since this is meant to give a flavour of the specifications.

```
/* Extracting Entities -                                          *
 *     Logical files accessed in programs are entities            *
 * Extracting Attributes -                                        *
 * Simple attributes are extracted from the record layouts */
```

//entity = ⟨ name, username, set of attributes, 01 level var,
// corresponding logicalfile, set of physical files ⟩
typedef entity **record** ⟨ **string**, **string**, **set** attribute, **decl**, **lfile**, **set pfile** ⟩;
//attribute = ⟨ name, username, type, var, property (PK/FK) ⟩

```
typedef attribute record ⟨ string, string, string, decl, string ⟩;
//Extract all entities and attributes of each entity
set entity getAllEntities ( )
{
  set lfile lFileSet;
  lfile lFile;
  set decl level01VarSet;
  decl level01var;
  set record ⟨ decl, string, lfile, set pfile ⟩ tempEntitySet = { };
  record ⟨ decl, string, lfile, set pfile ⟩ tempEntityTup;
  set attribute attrSet;
  set decl attribVarSet;
  decl attribVar; string assgnName, Uname, pic, entityName;
  entity e; set entity enSet = { };

  lFileSet = select $lfile from $PROAXROOT;
  for each lFile in lFileSet
  {
    level01VarSet = layout(lFile);
    for each level01Var in level0VarSet
    {
      tempEntitySet += ⟨ level01Var, assgnName, lFile, { } ⟩;
    }
  }
  for each tempEntityTup in tempEntitySet
  {
    level01Var = atIndex ( tempEntityTup, 0 );
    attribVarSet = getAttributes ( level01Var );
    for each attribVar in attribVarSet
    {
      Uname = getName ( attribVar );
      pic = getPicture ( attribVar );
      attrSet += ⟨ Uname, "null", pic, attribVar, "null" ⟩;
    }
    entityName = getName ( level01Var );
    lFile = atIndex ( tempEntityTup, 2 );
    aPfileSet = atIndex ( tempEntityTup, 3 );
    e = ⟨ entityName, "null", attrSet, level01Var, lFile, aPfileSet ⟩;
    enSet += e;
  }
```

```
  return enSet;
}
```

9 Conclusions

Based on our experience of building restructuring tools, we have evolved a specification driven reverse engineering framework that is independent of the programming language. The prototype has enabled tool builders to rapidly experiment with program analysis based, reverse engineering tools, and change the tool behaviour quickly. We believe that support for model generation, a proper balance between number of primitives and flexibility of specifications, and ability to integrate existing viewing tools, will go a long way to prove utility of this framework.

References

[1] SEEC Mosaic, URL: http://www.asera.com/seecmosaic.shtml.

[2] CERTC, Cobol Explorer from Reliance Technology Consultants, URL: http://www.cobolexplorer.com.

[3] Acacia, C++ Information Abstraction System, URL: http://www.research.att.com/sw/tools/Acacia/.

[4] CPPAnal and GUPRO, URL: http://www.uni-koblenz.de/ clange/IWPCPaper.pdf.

[5] Dali, Architecture Reconstruction Workbench, URL: http://www.sei.cmu.edu/ata/products-services/dali.html.

[6] Darpan: Tata Research Development and Design Centre, TRDDC, Pune, India: Internal Technical Report, US Patent Pending, URL: http://www.pune.tcs.co.in/aboutus.htm.

[7] Programmable Reverse Engineering: Scott R. Tilley, Kenny Wong, Margaret-Anne D. Storey, Hausi A. Muller, URL: http://www.rigi.csc.uvic.ca/Pages/publications/pre-abs.html.

[8] Graph Exchange Language: Richard C. Holt, Andy Schurr, Suson Elliot Sim, Andreas Winter, URL: http://www.gupro.de/GXL/Introduction/background.html.

[9] Revised Submission for MOF 2.0 Query / Views / Transformations RFP, URL: http://qvtp.org/downloads/.

[10] Model-based Design of Reverse Engineering Tools: Stan Jarzabek, Guosheng Wang: Journal of Software Maintenance: Research and Practice 1998.