St1-PL/1

Extracting quality information from PL/1 legacy ecosystems

Masterarbeit der Philosophisch-naturwissenschaftlichen Fakultät der
Universität Bern

vorgelegt von

Erik Aeschlimann

2014

Leiter der Arbeit:
Prof. Dr. Oscar Nierstrasz
Dr. Mircea Lungu
Institut für Informatik und angewandte Mathematik

Abstract

This thesis presents a case study of analyzing a legacy PL/1 ecosystem that has grown for 40 years and runs on a modern IBM mainframe. To support the stakeholders in analyzing it, we developed St1-PL/1 – a tool that parses the code for association data and computes structural metrics which it then visualizes using top down interactive exploration. Before building the tool and after demonstrating it to stakeholders we conducted several interviews to learn about legacy ecosystem analysis requirements.

We briefly introduce the tool, then present preliminary results of analyzing a large legacy ecosystem case study. We show that although the vision for the future is to have as much a decoupled architecture as possible the current state of the ecosystem is far from this. We also present some of the lessons learned during our experience discussions with stakeholders which include their interest in automatically assessing the quality of the legacy code.

Contents

1 Introduction

1.1 Overview

This master thesis

1. presents requirements for analysis tools aimed at large-scale industrial legacy ecosystems.
2. introduces a tool that supports the analysis and visualization of PL/1 ecosystems which conforms to the classical extract-abstract-view reference architecture.
3. analyzes a real-world large-scale legacy ecosystem and provides insight into the magnitude, complexity, and associated problems.

1.2 Personal Motivation

Fifteen years ago I decided to make an education in a company to get into the area of information technology. In this time a lot of big companies offered education-programs to help career changers to enter into the world of information technology. I began such a program to get educated as a software engineer for PL/1 in a mainframe environment. These programs were offered because there were not enough PL/1 programmers at this time so the companies had to educate them on their own. During the years of working on projects located on the mainframe I performed most of the roles within the development teams:

- Software engineer
- Software architect
- Application owner
- Requirement engineer
- Test manager

So I had the opportunity to look into a lot of different fields with different challenges and I saw that while performing all of the previous functions, I missed a lot of supporting technologies.

When I had to decide on the subject for my a master thesis, it looked obvious to combine my experience of working in a large company using a long grown legacy PL/1 ecosystem and the intention of doing analysis in the world of software ecosystems. And when Mircea Lungu came up with the idea of writing a tool in Smalltalk for analyzing the mainframe PL/1 ecosystem of my company, I was all for it.

I thought of possible ways to help the PL/1 programmers working on their applications. Because a lot of times I missed helpful tools supporting me in analyzing PL/1 code, it could be possible for me to provide my experiences to find an approach in defining how a new tool could look like and what information about the ecosystem it could deliver.

Furthermore the perspective to learn a new programming language, in this case Smalltalk, led me to implement a tool for analyzing large PL/1 ecosystems in Smalltalk.

1.3 Goals

After some discussions we found the main goals in a short time. We realized that there could be a need for new supporting tools in the process of software developing in the company I am working. And because it seemed that there is no tool that is able to visualize a complete PL/1 ecosystem, we wanted to develop such a tool in order to help the user in analyzing this ecosystem.

To have requirements for the implementation, we had to evaluate the needs of the employees. As a first step we interviewed employees to identify the needs and wishes of the different roles.

For the developing part of the master project we came with the following goals:

- It should be a basic system that shows the whole ecosystem in a single picture with some metrics and relationships.
- As a base for the visualization we wanted to take the structure the company had defined to group its business parts. Because the company does not have a model of the whole ecosystem yet, another goal of the thesis was to define a Meta-Model that is capable to specify the ecosystem from high level down to program level.
- A selection of what should be visualized and an interactive navigation between levels should also be implemented.
- In a first version the tool should be simple but already capable of delivering some results the company can use for analysis purposes. Other features would be implemented in further versions.
- In order that the work on this tool would not be in a rather theoretical way and the results could be used for evaluating the system, we also wanted to elaborate significant numbers for discussion purposes.

1.4 Technical

From the technical side there were not so many decisions to make. The team which would coach me during the work, is specialized in developing tools in Smalltalk, so it was clear that the tool would be implemented in this programming language.

For the analysis we decided to have clearly defined boundaries. So we wanted to focus on PL/1 programs running on the mainframe. And from these programs we wanted to extract some metrics and connection information. All the other parts, including Java, CORBA, Web Service and DB2 we did not look at. The reasons for this decision were:

- We could focus on one programming language and had to write only one parser. And this parser would be much simpler than one that has to extract information about all other possibilities of connections to different technologies.
- It was easier to achieve only the PL/1 part of the code than trying to get all kinds of code that are used in the company.
- The analysis would be easier when focusing on the PL/1 part.

As the development environment we have chosen Pharo / Moose. For parsing data out of the source code, like metrics and relations, we took the Smalltalk framework PetitParser and for the visualization part we used Quicksilver / Mondrian.

1.5 Results

During the work on the thesis we achieved following results:

- We defined a meta model to represent the structure of the ecosystem.
- We defined metrics we could get out of the source code.
- We implemented a parser for PL/1 programs to extract these metrics.
- We created a new tool called ST/1-PL/1 which is able to visualize the metrics and connections within Mondrian.
- We made some statistical evaluations with R to get interpretations of the quality of the analyzed code.
- Lots of the results are on the understanding level. We made a lot of experiences about how legacy ecosystems are maintained in large companies.

1.6 The Structure of the Thesis

After the short listing of some related work concerning this thesis in Chapter 2 we discuss in Chapter 3 the basic problem that led to the idea of this thesis. Within this chapter a short overview of large PL/1 ecosystems and the programming language PL/1 is given. The method of gathering the requirements for the planned tool is also explained. In Chapter 4 the tool and the parts of the architecture are presented, namely the meta model, the parser, the visualization and the metrics.

Chapter 5 presents the ecosystem that we examine with the tool and as a result a high level visualization of the entire ecosystem and three detail visualizations are shown with some discussion and in Chapter 6 a graph-theoretic analysis of the ecosystem is done. The last Chapter 7 discusses what experts thinks of our tool and also lessons learned and a conclusion is presented.

2
Related Work

The work presented in this thesis is related to different parts in software engineering. First it covers some parts of the reverse engineering, second it treats large scale software analysis and third the part of ecosystem visualization is covered.

2.1 Reverse Engineering

The general context in which the tool is used is reengineering. There is a rich literature on clustering in the reverse engineering community [27]. Thus the work of Demeyer *et al.* [4] is highly relevant even if it is targeted at analyzing OO systems. An even more relevant work is the approach presented by DeLucia *et al.* presented to migrating legacy systems towards OO systems [2]. The work on object oriented reengineering patterns of Demeyer *et al.* [4] is also relevant to our approach.

The idea of automatically aggregating dependencies from the lower level artefacts was first used by Muller in Rigi [17]. Rigi visualizes the data as hierarchical typed graphs and provides a Tcl interpreter for manipulating the graph data. The reconstruction process is based on a bottom-up process of grouping software elements into clusters by manually selecting the nodes and collapsing them. The same idea was then taken in other tools such as Shrimp [25].

By combining polymetric views [10] with treemaps our visualization can convey the same type of information as CodeCity [28]. However, the advantage of our visualization is that it provides information also about dependencies.

One ingenious mode of visualizing relationships between nodes organized in a hierarchy is that of Cornelisen *et al.* who use a circular bundle view that projects the system's structure in terms of hierarchical elements and relationships on a circle [1]. We consider using such a visualization in the future.

2.2 Software Ecosystems

There were several statistical case studies that observed the behavior of the developers working on a ecosystem.

Robbes [23] has surveyed the propagation of changes done within an API to the consumers of it, so called the ripple effect. It was a statistical study of two big software ecosystems, Squeak and Pharo, that treats the social effects within ecosystems. He proposed to use a lightweight model to analyze the evolution of an ecosystem. The results showed that lots of the developers did not follow the recommendations in the message explaining the changes.

Haenni [7] interviewed 20 developers about their needs of information for working with API. She distinguished the results of the interviews between upstream and downstream development.

Lungu has developed a tool for visualizing the evolution of object oriented software ecosystems and project repositories that we dubbed the Small Project Observatory [13]. The infrastructure in this thesis is targeted towards procedural languages instead, and does not leverage evolutionary information which was not available.

Ossher *et al.* resolve dependencies between projects in order to obtain a successful build of a target project in a Java repository of systems [19]. In our previous work we analyze and detect static relationships between the systems in a Smalltalk ecosystem [14].

Schenk [24] developed an API to visualize large scaled hierarchical graphs. The visualization includes the possibility to show weighted connections between the entities and also several metrics can be shown simultaneously. This API was used to implement the tool discussed in this thesis.

2.3 PL/1 Analysis Tools

The company that provided us with the case study uses IBM specific tools for their PL/1 development. XREF, a cross referencing tool, for example provides a list of programs that are affected by changes done in a subprogram (an external subroutine). This is very important to have all programs running in the production system with the same version of the subprogram. Otherwise for example the complex interaction between programs, DB2-plans, DB2 system and tables would not execute properly. This tool does not deliver complete information about the organization of the listed programs. The returned program-list only contains the data needed to inform the owners of the affected programs. It does not provide any kind of visualization or the possibility to navigate within the result.

Another tool used in the company is XINFO. With this tool it is possible to analyze the flow of the jobs, written in JCL (Job Control Language). It can list the dependencies between the jobs on level scheduling conditions and also the flow of in- and output files. The PC-Version of XINFO delivers a basic static visualization of the flow.

During the last years the company tried to introduce third party tools to help the engineers in analyzing the PL/1 code in an interactive way. But they did not succeed due to different reasons. On the one hand they were complex or not very intuitive in the handling, on the other hand the users were not educated enough. So the tools were not used and the further improvement was omitted.

But recently, Panorama[1], a third-party application that computes many metrics of the ecosystem and also creates visualizations, has been introduced in the company. Its main usage is in searching the code base with a short response time. At the moment it looks like this tool is accepted by the employees and will have a future in the company.

Neither Panorama nor the other tools offer the top-down approach we are working on, and we envision our tool will be used as a complement to it.

The operating system on the mainframe provides some searching tools, which are very efficient in searching a large amount of text. But the results of these tools are only text-based and there is no possibility to visualize them.

[1] http://www.itp-panorama.com

3 Problem

Large legacy ecosystems are a problem for a lot of companies that began to use information technology already 40 years ago. In this chapter we look at the problems that are based on the historical evolution of the ecosystem.

In the first section of this chapter a short overview over the procedural language PL/1 is given by looking at the parts that are relevant for this thesis. Afterwards we are looking at large legacy ecosystems and in the end the process of requirements gathering is explained.

3.1 A brief Overview of PL/1

PL/1, an acronym for Programming Language One, is a third-generation procedural programming language which was created in the 1970's by IBM. It combines features of Fortran and Cobol into a single language. Fortran was chosen due to its strength for scientific computing and Cobol for its qualities with respect to building business applications.

Figure 3.1 shows two typical PL/1 fragments. The first part PGM001.PGM represents a short PL/1 main program and the second VARIABLE.INCL is a code snippet that will be included into PGM001.PGM during the precompiler step. Because PL/1 was developed in the days of punched cards there are still limitations concerning the layout of the code. Each line of code is at most 80 characters long and only positions 2 to 73 are considered by the compiler. The first position is used for steering-signs for the printer and the last 8 positions are used to assign line numbers to the code.

Compiling a PL/1 program contains following four central steps:

- In the precompiler step, the code snippets are included into the source code and the precompiler statements are translated into PL/1 statements the main compiler understands. Also the SQL code is transformed to a form that is readable by the machine
- The compiler step then compiles the precompiled source code.
- During the link step the compiled mainprogram and the load modules of the statically called subprograms are linked together to a single load module.

- For programs using DB2 tables a bind step is needed, where the access path to the DB2 data is created.

PGM001.PGM

```
PGM001:   PROCEDURE OPTIONS (MAIN);                          001
                                                             002
      /* This program writes 'HELLO WORLD!' */               003
                                                             004
      DCL    FLAG      BIT(1) INIT('1'b);                    005
      DCL    %include VARIABLE;;                             006
                                                             007
      GOTO LOOP1;                                            008
                                                             009
      PUT SKIP DATA(TEXT);                                   010
                                                             011
LOOP1:   DO WHILE (FLAG = '1'b);                             012
            PUT SKIP DATA(TEXT);                             013
            FLAG = '0'b;                                     014
         END LOOP1;                                          015
                                                             016
      CALL EXTERNAL_PROCEDURE;                               017
                                                             018
END PGM001;                                                  019
```

VARIABLE.INCL

```
TEXT     CHAR(12)  INIT('HELLO WORLD!')                      901
```

Figure 3.1: PL/1 program

Certain aspects of PL/1 are important to understand later sections of this thesis:

- *Procedures.* Line 001 of Figure 3.1 uses the procedure statement and the keyword ‘MAIN’ to define this file as a main program. There are three types of code files: Main programs are able to run on their own. Subprograms are external procedures that can only be called by another program. Include files contain code fragments that are inserted into the program files during a preprocessing step. The same statement is also used within the program to define internal procedures, but these can be distinguished by the different keywords.
- *Comments.* Line 003 shows a PL/1 comment that has the same syntax as a C or Java one. The old development environment on the mainframe does not have a possibility to deliver some information of a program file that is stored somewhere outside the file. So in addition to documenting the intent of a program, it is important to have comments in the code that communicate information that cannot easily be read in the code, like external programs, used files or used databases. The new development environment supports the information of programs better, but it is still essential that a well structured comment is available in the code.
- *Variable Declarations.* Line 5 shows a simple variable declaration. It is a Bit variable that can store a single bit and therefore it can hold either ’1’B or ’0’B. These bit variables are often used as flags to control the flow of a program. Because the length of a line is limited in PL/1, there are a lot of variable names in the source code base that are shorter than four characters. Furthermore, in early systems, scrolling from one page to another could take several seconds, so engineers were encouraged to put as much information as possible into one page and so the variable names became very short. Therefore a lot of programs are very hard to read if the system of names is not known.

- *Includes.* Line 006 illustrates the include preprocessor statement. During preprocessing the code fragment from file VARIABLE.INCL replaces the include statement. The code fragment in the include file does not have to be a complete PL/1 statement. The main usage of this statement is to include the declaration of a data structure. This is needed for example when more than one program reads the same file. To assure that all of these programs read the file with the latest structure, the declaration of the structure is defined within an include file that is included by all of the reading programs.
- *Gotos.* Line 008 shows the goto statement used to jump to a label within the code. Although gotos were heavily used in legacy PL/1 code, they are considered bad practice today [5]. The Goto statement is one of the reasons that old programs do not have a well structured and readable flow, which we call spaghetti code today.
- *Calls.* Line 017 illustrates the call statement used in PL/1 to invoke external programs. There are two possible kinds of subprograms that can be called. On the one hand there are the statically called modules that are linked into the load module during the compilation of the main program. This has the drawback that a main program has to be recompiled every time a statically called subprogram has changed. On the other hand there are the dynamically called subprograms in which the linking is done during run time.

The PL/1 compiler received a major revision seven years ago with no backward compatibility and IBM intended to stop the support of the older versions of the compiler, so the company had to migrate the complete PL/1 codebase to the new compiler version. Because there was the need for a lot of programmers for a limited duration to do this migration, the company did this migration with the help of offshore companies. This migration was also used to do some redesign, like removing assembler subroutines for date/time conversion as well as minor code redesign (e.g. goto replacement). But there are still some program crashes due to not found incompatibilities and differences of behavior of the various versions of the compiler.

3.2 Large PL1 Ecosystems

Software systems do not exist in isolation without any connection to other systems, but are rather parts of *software ecosystems* in which multiple software systems interact with and depend on one another[12]. These subsystem may be implemented with different technologies and are running on any kind of machines like mainframe, servers, personal machines and even on mobile devices.

Consider the following example. At the end of every month a company has to send account information to all its customer. During this process several systems implemented and maintained by different teams are involved:

1. The first system has to collect all data that shall be sent at this moment to the customers and has to format it in a way the next system can use.
2. Because the data needed is distributed on various other systems, these system are also involved in delivering the data of the customer.
3. Then a system is needed that handles the information of what parts of the data has to be sent to the customer.
4. The next system prints the data in a defined format onto paper.
5. The last system involved in this process manages the shipping of the letters.

Most of the existing research into static analysis of software ecosystems has focused on open source software. Large commercial software systems have rarely been studied. For the company discussed in this thesis, I did not find an earlier analysis concerning the whole PL/1 ecosystem from a high level.

Software ecosystems of large enterprises often grow over many decades. The design of these ecosystems therefore reflects a mixture of programming styles and technologies that were used during these decades. Although some stakeholders would like to replace a systems after about 10 to 15 years of operation, there are still a lot of older systems running in production.

3.3 Initial Requirement Gathering

The developers in the company were not satisfied with the coverage of the existing tools. Therefore we conducted several free form interviews with employees aiming to identify their needs and wishes concerning tool support for analyzing the PL/1 ecosystem. The interviewees were chosen in a way to cover several diverse roles in the company: domain architects, solution architects, requirement engineers, software engineers and software testers. These people have been working with the existing environment for years and are accustomed with the existing processes.

To keep the interviews short, only a small number of discussion points were defined for the questionnaire. In the following list the four questions with the answers are listed. Because the company uses other technologies than PL/1, all the questions are in relation to PL/I and its belonging parts.

1. *"What are the main activities you are executing concerning your role within the development teams?"*

    - Most of the asked people answered that their main activities related to the PL/1 code are to create, analyze and maintain programs.
    - Also, the analyzing and solving of problems and program crashes is one of the central work fields performed by the employees.
    - Creating test-cases, based on code comparison and requirement specification.
    - Managing requirements and mapping them onto applications.
    - Analyzing aspects of the ecosystems, for example: how is a application affected by another; how are the domains connected between each other, what affect does a change of a program have to other programs in other applications.
    - Taking measurements to optimize the performance of the programs.
    - Performing reverse engineering.
    - Checking if an application is still used (Domain Architect).

2. *"What specific information or data do you need to be able to perform these activities? These information do not have to be something technical as a file or program, it can be anything they need."*

    - The data most needed are the PL/I source code files.
    - A very important part is the knowledge, either one's own or that of colleagues.
    - The official PL/1 manuals offered by IBM are important reference books.
    - Information about the connection between programs.
    - Business cases are used for testing and reproducing program crashes.
    - Requirement specifications.

- Architecture documents.
- Business processes.

3. *"Do you use tools to support your work, for example to analyze the PL/1 code and the relationships within the large ecosystem? What kind of tools do you use? It does not have to be a tool directly associated with the PL/1 code on the mainframe."*

   - Software development tools (IBM RDz Workbench, an Eclipse clone for PL/1)
   - Software debugging tools
   - Software comparison tools
   - Job running tool
   - PL/I code reading tool
   - UML modeling tool
   - Plaintext search
   - Measurement tool
   - Error analyzing tool (Fault Analyzer)
   - Phone, get knowledge from another (very important)

4. *"Do you already have some wishes or needs concerning a new tool supporting their work?"*

   - Version comparing tool for all kind of files (e.g. for creating test cases)
   - Show indirect calls (MQ, files, DB's)
   - Better development environment (java like, better editor)
   - Overview over domains, cross-domain links (for discussions and strategy)
   - Code generator
   - Dynamic flow of data, input tracking
   - Diagram generating, for documentation purpose

From these interviews we extracted three requirements for the analysis tool we wanted to focus on:

R1. **Produce high-level views of the entire code base.**

Because there is no analysis tool that is capable of showing the ecosystem from a high level point, the architects directly read in the code to collect relevant information, even for high level analysis.

All of the interviewed employees use standard mainframe tools to analyze the code, like cross-reference, performance measuring or dump analysis tools. None of these tools feature graphically visualized output showing the big picture of the ecosystem.

However, when asked, the different respondents provided different reasons for which they need a big overview of the system: some architects want a high-level overview of the domains for discussion purposes. Some engineers need tools to create high-level diagrams for documentation purposes. Finally, testers need tools to identify parts of the system that should be tested more intensively.

R2. **Provide code quality information.**

The quality of the different parts of the codebase is a cross-cutting concern for the stakeholders. Again, just like in the case of the previous requirement, the rationale for the necessity of quality information is reported differently by different stakeholders: the testers want to optimize the focus of the tests; the architects want to know which parts of the system must be redesigned, and management needs to know how to allocate the restructuring effort.

R3. **Support ecosystem restructuring**.

Due to the lack of a global view on the PL/1 part of the ecosystem during the evolution of the ecosystem, strong coupling relationships evolved between the domains and subdomains.

A long-term goal of the company is to decouple the individual systems and sub-systems and increase the modularity of the ecosystem. This would simplify the replacement of whole subsystems with alternatives written in trending languages or third party products.

These requirements have not been corroborated with other companies and although we suspect that they are not specific to the company we have been discussing with or to PL/1 ecosystems, we have no proof otherwise. It remains for the future or for other researchers to gather requirements from other industrial companies.

Additional these requirements are quite specific to industry and legacy systems as another study that we have been conducting recently shows that developers in open source systems have different information needs, motivations, and goals [7].

4

St1-PL1 — A Visualizing Tool

To address the requirements presented in the previous section we created a tool called St1-PL/1 by customizing components of the Moose analysis framework [18].

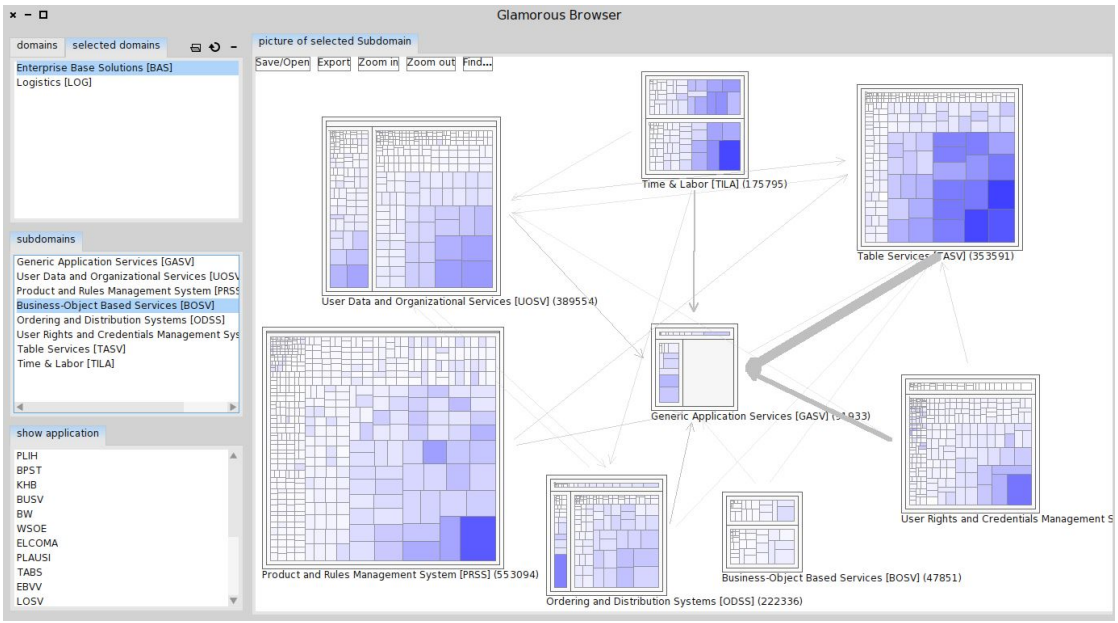

Figure 4.1: Picture of St1-PL/1

Figure 4.1 shows a picture of St1-PL/1. In the right part you see the visualization of a domain. Every box represents a subdomain within this domain and every square within a box represents an application or program within the subdomain. The size of the squares and the colors represents the metrics. The arrows

between the subdomains illustrate the number of call-connections between the subdomains. Details to this visualization are discussed in Chapter 5. On the left side you see lists that are used to define which parts of the ecosystem are shown in the visualization.

The architecture of the tool is an instance of the classical Extract-Abstract-View reverse engineering tool reference architecture [6]. Data about source code artifacts is extracted into a software repository and then using different analysis techniques, abstracted to provide a more condensed view of the analyzed system. Abstractions of the system are viewed by appropriate visualization means.

This chapter first shows the architecture of the tool, afterwards it explains the meta model used to structure the ecosystem. Then a deeper view into the parser is given and at the end the visualization part and the metrics that are used to measure the ecosystem are explained.

4.1 Architecture

In the following the individual components are listed with discussions of some particular choices made to adapt the infrastructure to our case study. We discuss in turn the following four components:

1. A meta-model which is an extension of FAMIX is populated with information extracted from the source code and other external sources of information.
2. A model extractor based on island grammars [16] is built with the PetitParser framework [22].
3. Graph analysis and visualization tools are built to work on top of this meta-model.
4. The Metrics implemented into the tool.

4.1.1 Meta-Model

The goal was to find a way to map the structure of the examined ecosystem to a model that has the following benefits:

- it should be a model that is already in use and can be used to compare our ecosystem to other already analyzed ecosystems, which does not have to be implemented by a procedural language. We thought it could be interesting to compare our procedural system to an Object oriented system.
- The model should represent the actual structure of the company that they have laid onto their ecosystem and describes the business structure behind all the applications of the ecosystem.
- Another very important quality of the model should be the possibility of using existing visualization frameworks already implemented in the developing environment.

In order to leverage the Moose analysis platform and the tools it offers, we extended FAMIX [26] (Moose's metamodel) with entities required for analyzing a PL/1 ecosystem. In particular we had to extend the original meta-model to be able to model the hierarchical organization of our subject ecosystem.

Figure 4.2 illustrates the extended meta-model. The elements in the figure are:

- FAMIX classes, shown in white. These classes are taken directly from the existing FAMIX Meta Model, because they also fit to our model in mind.
- PL/1 specific classes, in blue. When designing the tools we focused on metrics considering the programs and the call-connections between these programs. So we only took a very basic amount of classes to represent the language specific part.

- The classes colored in grey represent classes that are taken from the standard FAMIX model and to which we added some attributes needed for our analysis and visualization.
- The yellow colored entities represent Ecosystem-specific containers indicating the structural levels defined by the company. They are not PL/1 language specific but fit best to the scoping entities of the standard FAMIX model.

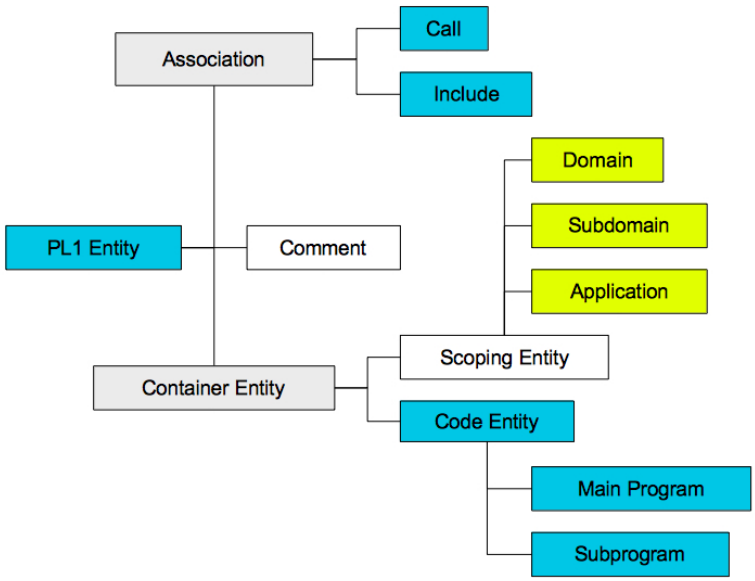

Figure 4.2: Meta-model of PL/1 Ecosystem

The meta-model is lightweight and was sufficient for our purposes. However, for more detailed analysis one can use one of the other existing PL/1 meta-models, such as the one of Hugo Bruneliere[1].

4.1.2 Parser

To analyze the ecosystem some information required by the meta-model was needed. This data had to be extracted directly out of the PL/1 code. And because no already parsed results of the entire PL/1 code base were available, where we could obtain the data, the implementation of an own parser was unavoidable.

To extract we had to parse source code as well as external sources of information like the ones mentioned in Chapter 3.

For parsing the code we used PetitParser, a parser framework based on PEGs (parsing expression grammars) [22]. Since we only need to identify specific code parts, we did not implement a complete PL/1 parser but we focused on the statements we needed by using an island parser [16].

As a first step the extracted connections only represent the fix coded program-to-program calls mentioned in Section 3.1. There are also not a lot of metrics that are evaluated during the parser step. So the parser could be reduced to a little number of rules:

- cutting out the comments

[1] http://www.emn.fr/z-info/atlanmod/index.php/Ecore

- counting the lines of code
- extract call statements
- extract goto statements
- distinguish between calls of inner procedures and calls of external programs

As a first example of a method implemented in PetitParser, Figure 4.3 shows the way the GOTO statements are counted for the GOTO metric. This method does no more than increasing the GOTO-counter by one, no object for the parsing result is created.

```
gotoStatement

    ^ (   'GOTO' asParser caseInsensitive trim,
          identifier,
          $; asParser negate star )

      ==> [:nodes| self numberOfGoto: ((self numberOfGoto) + 1) ]
```

Figure 4.3: The parser method for the goto statement

The next example, the call statement, is a central part of the analysis and visualization part, so we have a closer look at it. Figure 4.4 shows the method that is implemented in PetitParser to extract the call:

```
callStatement

    ^ (   'CALL' asParser caseInsensitive trim,
          identifier,
          $; asParser negate star )

      ==> [:nodes| self numberOfCall: self numberOfCall +1.
            PL1Call  new    caller: parsedFileName;
                            target: nodes second.
            ]
```

Figure 4.4: The parser method for the call statement

Because an inner program call looks the same as a call to an external module, all created call-entities have to be checked after parsing, if the identifier is an existing program. If there is no matching program to the identifier, the call entity is not taken into consideration.

Another problem with this approach is the possibility of dynamic calls. In PL/1 a call to another program can be implemented dynamically, where the name of the called program is defined during runtime. So it is not possible to extract any program name statically by parsing the code.

There are a lot of possible connections between programs that are not covered by the rules listed above. But for a first version of the tool and some first analysis, the inter-program-calls are sufficient. For future versions following technologies of passing data between programs should also be implemented into the tool.

- databases: Writing and reading a field of a table is a kind of data flow between programs.
- files: data is also passed on by writing and reading files.
- CORBA / Web Service: An important channel to pass data is the synchronal communication implemented with CORBA or Web Service that is used in the ecosystem mainly to connect the JAVA front-end with the PL/1 back-end. But also connections through this middleware from PL/1 to PL/1 are established.
- MQ Series: Passing the communication data by sending messages through a message-queueing system.

But to add these connections between programs to our tool, additional information that is not stored within the PL/1 code, is needed. For example to extract a connection by file, the JCL-Job-files that contain the physical information about the files, have to be parsed as well.

Another limitation to the parsed result is the two step compilation process in PL/1. In a first step only the preprocessor statements are compiled and the result is inserted into the source code. In the second step the whole sourcecode is compiled. Because we have limited our Case Study to the code basis before the precompiling step, there can be some call associations that are not found by the parser. in Figure 4.5 you see an example of such an association that is affected by the preprocessor and that is not found by the parser:

Definition of precompiler procedure CAL

```
%DCL CAL ENTRY
%CAL: PROC(IN) RETURNS (CHAR);
   DCL (IN, OUT) CHAR

   OUT = 'CALL ' !! IN;

   RETURN (OUT);

%END CAL;
```

Use of precompiler procedure CAL in source code

```
CAL (otherProc1);
```

Same statement after precompiler step

```
CALL otherProc1;
```

Figure 4.5: Example of the use of precompiler statements

But although there are limitations, which are listed above, the result gained out of the parsing step delivers a result that can be used for further analysis.

An external application on the mainframe manages the mapping between individual programs and the higher level abstractions in the ecosystem organization such as domains and subdomains. We thus extracted this information to produce the metadata needed for our approach from an external file that provides a list of all programs and their accompanying applications, subdomains and domains. Figure 4.6 shows a fragment of the extracted hierarchy data.

AA1000_PLB	ICTO-218	PL/I - Helper - PLIH	PLIH	Enterprise Base Solutions [BAS]	Generic Application Services [GASV]
AA2000_PLB	ICTO-218	PL/I - Helper - PLIH	PLIH	Enterprise Base Solutions [BAS]	Generic Application Services [GASV]
AA3000_PLB	ICTO-218	PL/I - Helper - PLIH	PLIH	Enterprise Base Solutions [BAS]	Generic Application Services [GASV]
YAAU1_PLU	ICTO-218	PL/I - Helper - PLIH	PLIH	Enterprise Base Solutions [BAS]	Generic Application Services [GASV]
YAAU2_PLU	ICTO-218	PL/I - Helper - PLIH	PLIH	Enterprise Base Solutions [BAS]	Generic Application Services [GASV]
YAAU3_PLU	ICTO-218	PL/I - Helper - PLIH	PLIH	Enterprise Base Solutions [BAS]	Generic Application Services [GASV]
EE1000_PLO	ICTO-4281	Business Unit Services	BUSV	Enterprise Base Solutions [BAS]	Business-Object Based Services [BOSV]
EE2000_PLO	ICTO-4281	Business Unit Services	BUSV	Enterprise Base Solutions [BAS]	Business-Object Based Services [BOSV]
EE3000_PLO	ICTO-4281	Business Unit Services	BUSV	Enterprise Base Solutions [BAS]	Business-Object Based Services [BOSV]
YEEU1_PLG	ICTO-4281	Business Unit Services	BUSV	Enterprise Base Solutions [BAS]	Business-Object Based Services [BOSV]
YEEU2_PLG	ICTO-4281	Business Unit Services	BUSV	Enterprise Base Solutions [BAS]	Business-Object Based Services [BOSV]
YEEU3_PLG	ICTO-4281	Business Unit Services	BUSV	Enterprise Base Solutions [BAS]	Business-Object Based Services [BOSV]
YEEU4_PLG	ICTO-4281	Business Unit Services	BUSV	Enterprise Base Solutions [BAS]	Business-Object Based Services [BOSV]
YEEU5_PLG	ICTO-4281	Business Unit Services	BUSV	Enterprise Base Solutions [BAS]	Business-Object Based Services [BOSV]

Figure 4.6: An external file contains the mapping between the programs and their external organization

4.1.3 Visualization

In order to integrate the tool into an existing developing environment, namely Moose, no own visualization part was implemented for the tool. So following frameworks were used to build the visualization:

- *Glamour* is a framework for building interactive browsers in an easy and fast way. It offers several window types that can be put together to an interactive user interface being able to visualize the data in several ways.
- *Quicksilver* is used in this thesis for two tasks.
  - to calculate the connections between the higher level entities out of the program connections.
  - to create the picture part of the visualization by arranging the entities in container entities and by coloring the entities defined by metrics given to the framework.

4.1.4 Metrics

To analyze the ecosystem some data is needed to be visualized in the tool. Beside the connection between the entities some metrics that could be helpful, were also defined. In the following these metrics are explained.

We compute metrics which can be derived from our meta-model but also retrieve metrics from other sources in the company. The metrics used in later parts of this article are:

1. LOC — the number of line breaks in an entity. Comments and empty lines therefore also affect the result.
2. FAN-IN — sums up all incoming calls of an entity. The source programs of these calls can be located in the same application but also in other applications located in the same subdomain or in different subdomains or domains.
3. FAN-OUT — sums up all outgoing calls of an entity. The target programs of these calls can be located in the same application but also in other applications located in the same subdomain or in different subdomains or domains.
4. GOTO — the number of goto statements in an entity. The usage of the statement is discouraged but they are known to still exist in the code.

5. FP — The number of function points in an entity. This number is calculated by weighting and summing up functionality in a program, like I/O's or program calls. We obtain this information from an external document available in the company.

To calculate metrics for enclosing entities, the tool sums up the values of all entities one level beneath them.

In this chapter the technical part of St1-PL1 was introduced. In Chapter 5 the functionality of the tool will be explained on the basis of a case study.

5 Case Study — Visualizations

The tool introduced in Chapter 4 is used in this chapter to analyze a specific legacy ecosystem. The first section presents the ecosystem and lists the technical details of it. Afterwards the ecosystem is analyzed by the usage of one high level and four detail level visualizations.

5.1 The examined Ecosystem

The ecosystem we analyze in this thesis, belongs to a company that started using information technology 40 years ago. This section introduces this ecosystem and explains what restrictions were made in this thesis.

The following technologies are used within the ecosystem:

1. Programming languages
   - PL/1: used in back-end and front-end, runs on Mainframe
   - JAVA: used for front-end GUI, runs on servers
2. Middleware
   - CORBA / Web Service: SOA approach, only PL/1 as provider of data
   - MQ Series: for message queueing
3. databases
   - DB2: relational database
   - IMS: hierarchical database, at end of life

Here are some numbers about the examined ecosystem. These numbers only cover the PL/1 part of the Swiss platform that runs on an IBM Mainframe.

1. Lines of Code

all programs	140'000'000
relevant line of codes	73'000'000
before precompiler step	30'000'000

2. files

main programs	13'000
sub-programs	19'000
JCL Jobs	43'000

3. databases

DB2 tables	80'000
IMS	500

The ecosystem consists of various applications written in various programming languages and interacting with each other through services, databases, queues, files and static dependencies.

The thesis covers the core of this legacy ecosystem written in PL/1, running on an modern IBM mainframe machine with a modern operating system. The main task of the PL/1 part is to support the backend business processes and to ensure the data storage and data processing. So the analysis within this thesis always considers a mono-lingual ecosystem. Approaches to program understanding for multi-language systems have been proposed before [9, 20].

The PL/1 source code base consists of 30 MLOC before preprocessing and 130 MLOC after. For performance reasons we limited our analysis to the source code before preprocessing.

There are only a few PL/1 applications which run isolated on the mainframe; the standard architecture is that at least the presentation layer and if possible even part of the business logic is implemented in Java and the interface to the mainframe modules is done via CORBA/Web service. But there are still a lot of applications that have PL/1 user interfaces that run on the IMS platform.

The individual components of the PL/1 ecosystem are applications which interact with each other through services and direct calls. They have a certain degree of independence in their evolution and are composed of multiple programs.

But only recently the company began to define an organization to structure their information technology system. This structure should mirror the existing business products and flows.

Given the ample number of applications and since only recent versions of the PL/1 compiler support structuring of programs into packages, developers had to devise ad hoc methods to structure the code base. Thus the applications have been categorized into *sub-domains* and *domains*. Each of these domains contains up to a dozen subdomains and each of these subdomains contains multiple applications that logically belong together.

The initial structure was created by analyzing the business processes and the applications were then distributed to the subdomains in a best possible way. Then during several years the structure was optimized by adding or removing new entities and moving applications to other entities if they didn't fit into the old one. Too big applications were also split into smaller ones and small applications were merged to bigger ones.

Figure 5.1 illustrates the four levels into which the source code is structured. At the top level there are about 30 *domains*. The domains and subdomains replace a package system which has only recently been incorporated in PL/1.

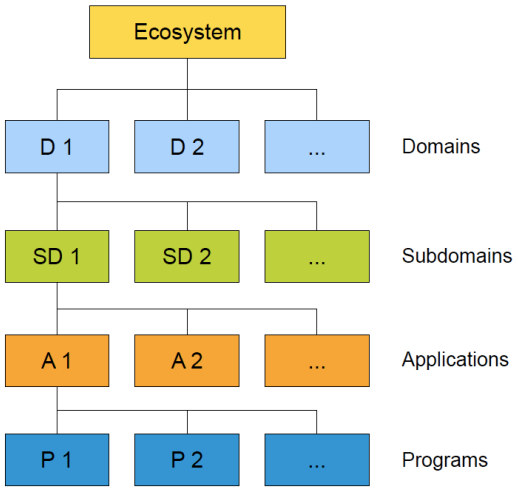

Figure 5.1: The applications in the ecosystem are organized in a hierarchy with four levels

5.2 High Level Visualizations

5.2.1 Top-Down Exploration

To begin our analysis we visualize the relationships between the domains and the amount of functionality in each of the domains (R1) with St1-PL/1. Figure 5.2 presents the overview of the domains and their relationships. Here we detail the main four of the construction principles:

- **Polymetric views.** Each of the 25 standalone squares represents an individual domain. The area of each square is proportional to the LOC of the domain as aggregated from the program level.
- **Dependency Graphs.** The arrows between the top-level squares represent the calls between the domains. The size of each arrow is proportional to the number of calls that are made between two corresponding domains.
- **Tree Maps.** The figures for each of the domains are drawn using a space-filling technique similar to tree-maps [8] which shows the contained entities all the way to the program level. The surfaces of these entities are also proportional to their corresponding LOC.
- **Highlighting.** The blue color-saturation of the individual programs is proportional to the number of contained function points. Darker colored programs have more function points than brighter colored ones. Since the human eye cannot distinguish between too many shades of blue, we map all the values to five distinct shades which correspond to quartiles.

Size and functionality The figure shows that the domains vary in size significantly from the smallest (N25), which has 17 KLOC, to the largest (N16), encompassing 4.6 MLOC.

Inside the domains, the applications are visible as clusters of programs. Some domains contain only one large application (*e.g.,* N24) while others contain a large number of applications (*e.g.,* N16). One can select an individual (domain, subdomain, application) and learn about its interaction with the rest of the ecosystem.

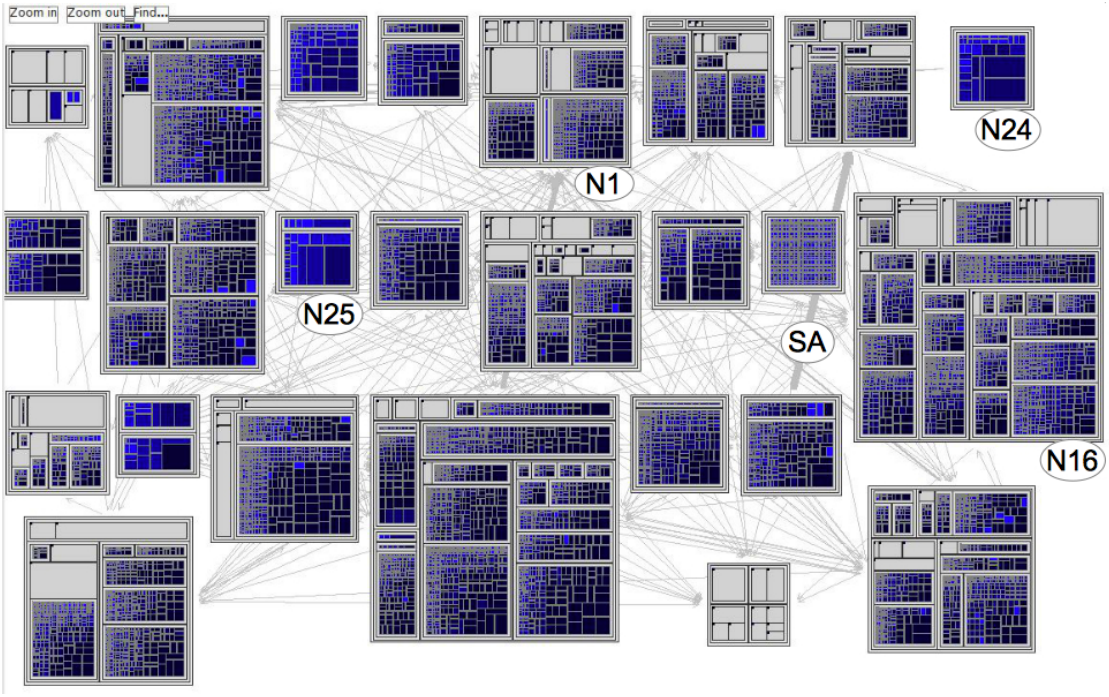

Figure 5.2: Visualization of the whole PL/1 ecosystem showing call relationships between domains and highlighting function points

Coupling The figure shows a large number of dependencies between the domains. Given the requirement for restructuring the ecosystem and decoupling the domains (R3) the figure hints at the challenges of the task ahead and at the need for an incremental strategy.

However, besides the general large number of dependencies the figure also shows that the number of connections between the domains varies widely: from domain N24 which is almost completely isolated to domain N1 which is strongly connected to the rest of the ecosystem [1].

The dependency marked with (SA) illustrates a strong relationship between the two associated domains. Using the tool one can inspect an individual dependency and learn the reasons for its existence: does it exist because of many calls to a narrow API or due to a large palette of exposed functions?

[1] By applying the page rank algorithm on the domain dependency graph we learn that domain N1 ranks the highest. We validate with one of the stakeholders that the domain actually is a critical one in the company.

5.3 Four Visualizations

Given the importance of domain N1 we decide to *zoom in* and learn about the quality of the code inside it (R2). So in the next four subsections this domain is visualized on level subdomain with three different metrics.

5.3.1 Function Points

In Figure 5.3 the number of function points of each program is shown. In opposition to Figure 5.2 the saturation of the color in this figure is linear to the number of function points. As mentioned in Subsection 4.1.4 the tool does not calculate or extract these metrics out of the source-code itself. The values are taken from a file extracted from the Panorama tool used in the company. The function point values are not parsed out of the code by Panorama but are calculated by a formula which counts the different kinds of weighted interactions of a program.

Figure 5.3: Visualizing the functional points of code inside the domain N1

There are some observations we can see in the figure:

- Big programs have more function points than small programs.
- In this figure we cannot locate an interesting program that does not fit in the pattern mentioned in point one. This leads to the question, whether the approach of calculating function points rather than parsing them leads to a satisfactory result.

- There is another relationship that can be observed within this picture: Because the number of program calls flows into the calculation of the number of function points, the biggest arrow starts at the subdomain marked with (G) with the most function points.
  But because other functionalities and calls to programs in the same applications or in other domains are considered for calculating the number of function points, this relationship is not very useful for analyzing the ecosystem and does not help to get more information out of the ecosystem.

5.3.2 GOTO

As one of the aspects of quality we check compliance with coding style guidelines, particularly the usage of GOTO instructions which the guidelines strongly discourage. In Figure 5.4, every program that contains GOTO statements is highlighted in blue.

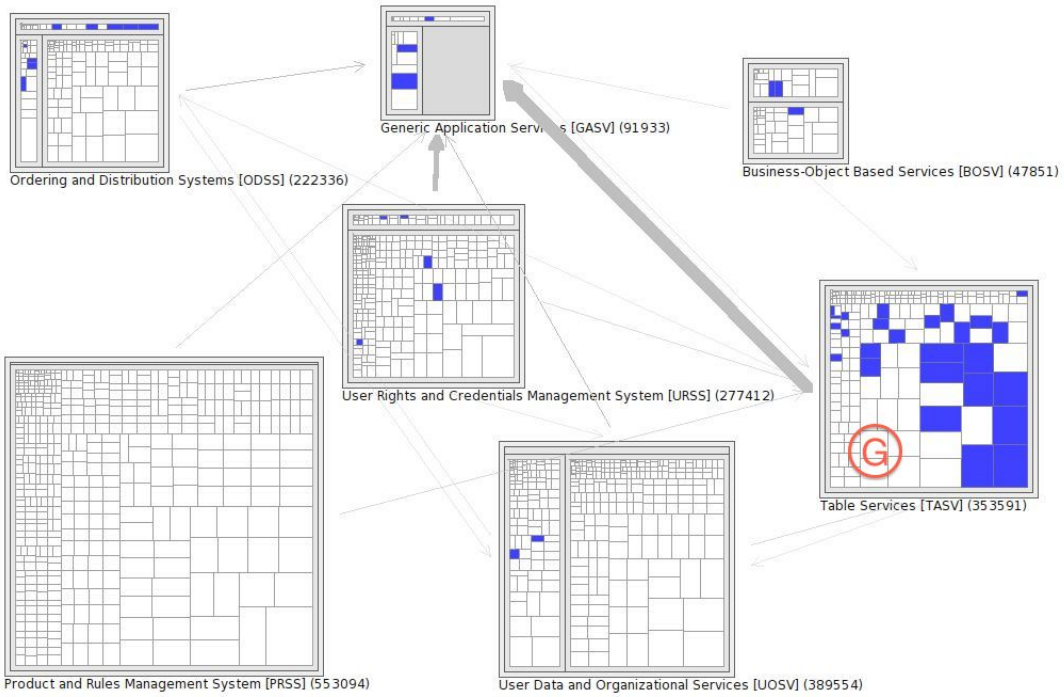

Figure 5.4: Visualizing the usage of GOTO inside the domain N1

Some interesting points we can see in the figure:

- One can see clearly that the subdomain [TASV] marked with (G) at the right border could be a candidate for reengineering, because the presence of GOTO's is a sign of an antiquated programming style.
- Except of the subdomain [TASV] the domain looks good overall and does only need minor effort to remove the GOTO's from the code.

5.3.3 Fan-In

The third detailed visualization treats the incoming connections, fan-in. Figure 5.5 visualizes the fan-in in a similar way to the GOTO metric in Subsection 5.3.2. Every program that has more than 10 incoming connections is colored blue.

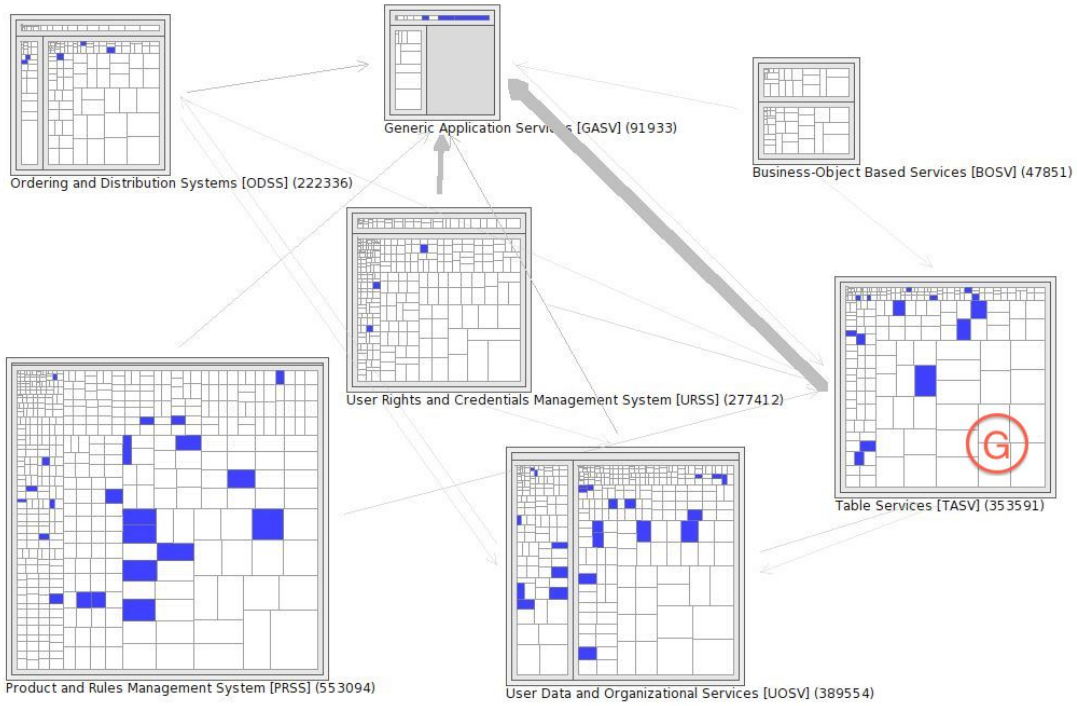

Figure 5.5: Visualizing Fan-In >10 inside the domain N1

There are some observations we can draw in the figure:

- With Figure 5.5 it is possible to identify subdomains that offer functionalities to programs inside or outside itself and therefore are more delicate for code changes.
- This visualization is helpful to identify quickly the subdomains with many connections within a domain. But it does not say anything about the number of connections. So for example there is a program in [TASV] marked with (G) that manages the access onto central tables and that is called more than 1000 times.

5.3.4 Fan-Out

The last Figure 5.5 visualizes the fan-out in a similar way than the metrics presented in Subsection 5.3.2 and Subsection 5.3.3 . Figure 5.6, visualizing the fan-out metric, has a threshold of 10 outgoing connections for coloring the program blue.

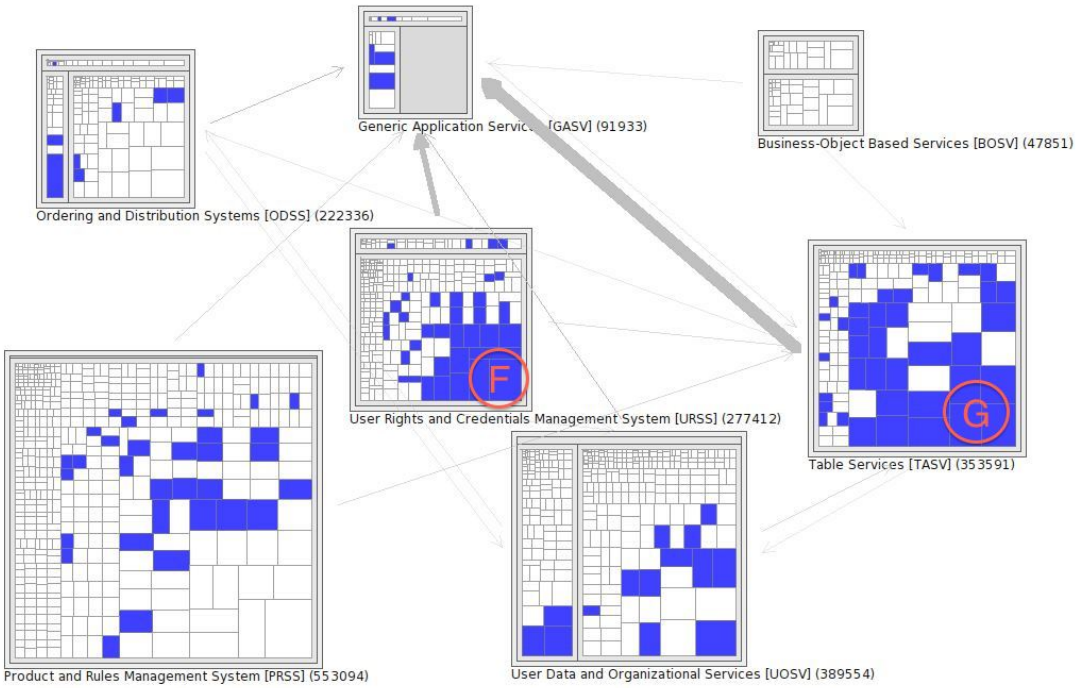

Figure 5.6: Visualizing Fan-Out >10 inside the domain N1

There are some observations we can see in the figure:

- The number of blue programs in an application in Figure 5.6 correlates with the sizes of the arrows starting at this subdomain. For example the two subdomains with the most blue programs, [URSS] marked with (F) and [TASV] marked with (G), are also the start-point of the two biggest arrows in the figure.
- This visualization helps to identify subdomains with many outgoing connections.

6 Case Study — Graph-Theoretic Analysis

The main function of St1-PL/1 is to search and identify problematic parts of the ecosystem in an interactive way. By zooming in from the top level of the ecosystem to lower levels, the user can isolate the applications where reengineering would be required. But this approach is rather an interpretation than based on strong scientific facts.

So we needed another way to find meaningful values which could complement the description of our interactive analysis and could help to identify the important nodes within the ecosystem. The solution was to process some graph-theoretic analysis of the ecosystem facilitated by St1-PL/1. Graph-theoretic analysis has been used in software analysis for other purposes including predicting modules which are prone to have defects in the future [29]. In our case we use it to better understand the nature of the ecosystem case study. The reported results will be of interest and a reference point to researchers and managers of large legacy ecosystems.

Another benefit of the measurements created in this chapter could be a comparison of this legacy ecosystem with open-source ecosystems in a future work.

6.1 Resources

To carry out the analysis we used the programming language R that was developed in the 90's as a part of the GNU-Project. R followed the programming language S that was developed to compute statistical data, but R was open and free to use. R can be easily expanded by packages. For our graph-theoretic analysis in this thesis we used the packages *statnet* and *igraph*. Statnet offers the methods to compute the various metrics such as centrality or degree that are needed for the computation of the graph-theoretic analysis. The package igraph contains the methods we used for the visualization of the results.

6.2 Things to measure

For the measurement we had to create a network that could be analyzed with R. For this purpose we extracted all calls that were identified by the parser within the ecosystem out of the tool. These calls can be loaded as a simple list into R and every call is then represented as an edge of the graph. The nodes of the

graph are then derived out of the edges. As a result we got directed graphs on level domain, subdomain, application and program.

Because the nodes that do not have any connection to other nodes would not be part of the graph, this method does not exactly reproduce the ecosystem. But for the purpose of finding critical parts of the ecosystem, the entities without connection to any other entity were not interesting at all. So our graphs were sufficient for the analysis.

6.3 Graph Theory Metrics

Before showing the results of the analysis in Section 6.4 and Section 6.5 the metrics used within these sections and their benefit have to be described:

- Centrality — The centrality of a node measures the importance of it within a graph. Therefore centrality can be used to identify important nodes within a graph. In our case study the centrality helps to identify the important parts of the ecosystem that are candidates for closer examination. Centrality is not a metric on its own, but rather a group of metrics. In this thesis following centrality metrics are used:
  - Degree — The degree of a node is defined by the sum of all incoming and outgoing connections. On program level these connections are the calls between programs. On the other levels the connections are the sum of the calls from all programs of one entity to the programs of another entity. Degree = indegree + outdegree.
  - Indegree — For the indegree metric only the incoming connections are considered.
  - Outdegree — For the outdegree metric only the outgoing connections are considered.
  - PageRank — PageRank is not one of the classical centrality metrics. It was developed to measure the linking between documents within the World Wide Web. PageRank calculates the importance of a node by summing-up the weights of the edges that links to it. The weight of an edge depends on the PageRank of its source node. The sum of all PageRanks within a graph is equal to 1.
- Density — The density of a graph is defined as the ratio between the existing number of edges and the maximal possible number of edges. Its value ranges from 0 to 1, where the complete graph has a density of 1. With this metric it is possible to see how strong the entities in the ecosystem are connected. It cannot be used to identify single entities with special character.

6.4 Network Level

We start with computing the density of each of the graphs. This number helps us to characterize the graphs in the context of how many connections there are proportionally to the number of edges. This number shows us how complex the nodes are connected on each level of the ecosystem.

Table 6.1 shows a high density at the domain and subdomain levels. At the application level where the basic units of our ecosystem live we observe a density of 6%. When we take into account that there are 229 applications we can calculate that every applications is connected to 7 other applications on average. This is not a bad value when we see that a lot of actions within a bank application needs a lot of data from other applications. So we can say the encapsulation is on a good level.

Level	Node Count	Density
Domain	25	0.463
Subdomain	75	0.208
Application	229	0.06
Program	16819	0.000189

Table 6.1: Indices gained on graph Level (GLI's)

6.5 Node Level

Information about the relative importance of nodes and edges in a graph is computed through centrality metrics. Previous work in reverse engineering has shown that centrality measures can support detecting the critical software components [21].

We compute the centrality of the nodes at the different abstraction levels in the ecosystem with four different measures: Degree centrality, indegree, outdegree, and PageRank.

For this graph-theoretic analysis only unweighted connections were used, so for the result it was not considered, how many calls there are between two entities.

	N1	N2	N3	N4	N5	N6	N7	N8	N9
PR	0.08	0.067	0.06	0.057	0.057	0.056	0.054	0.05	0.05
Deg	35	36	19	34	18	17	32	30	30
Out	15	20	0	17	0	0	16	14	15
In	20	16	19	17	18	17	16	16	15

	N10	N11	N12	N13	N14	N15	N16	N17
PR	0.05	0.047	0.04	0.039	0.038	0.037	0.034	0.03
Deg	28	29	24	12	22	17	23	25
Out	14	15	13	2	12	6	15	18
In	14	14	11	10	10	11	8	7

	N18	N19	N20	N21	N22	N23	N24	N25
PR	0.03	0.025	0.02	0.018	0.012	0.012	0.012	0.012
Deg	17	22	11	16	9	4	2	0
Out	12	17	7	13	9	4	2	0
In	5	5	4	3	0	0	0	0

Table 6.2: Indices gained on level domain

Table 6.2 shows these measurements for the domain dependency network. The nodes are ranked in decreasing order of their importance as measured by their PageRank. We see several domains with a high PageRank.

For example domain N1 (also highlighted in Figure 5.2 and discussed earlier) has the highest PageRank and the highest indegree. When inspecting this domain, we see that it contains infrastructure systems that provide basic functions to numerous other systems in all the domains, therefore it has a larger centrality than the other domains. So analyzing the centrality metrics helped us to identify a domain important for other applications.

What we also see is that the indegree is very similar to the PageRank. When sorting by the indegree only three nodes (N2, N5 and N15) would change their rank.

Figure 6.1 to Figure 6.3 show the distribution of centrality in the ecosystem at the subdomain, application and program level. We see that as the number of vertices in the graph increases (as listed in Table 6.1) the degree spectrum also increases.

Because the subdomain graph has a higher density than the application and program graphs, it also has more connections per node on average. Thus the range of degrees is much wider than in the other graphs that have a lower density and where most of the nodes are accumulated at the lower end.

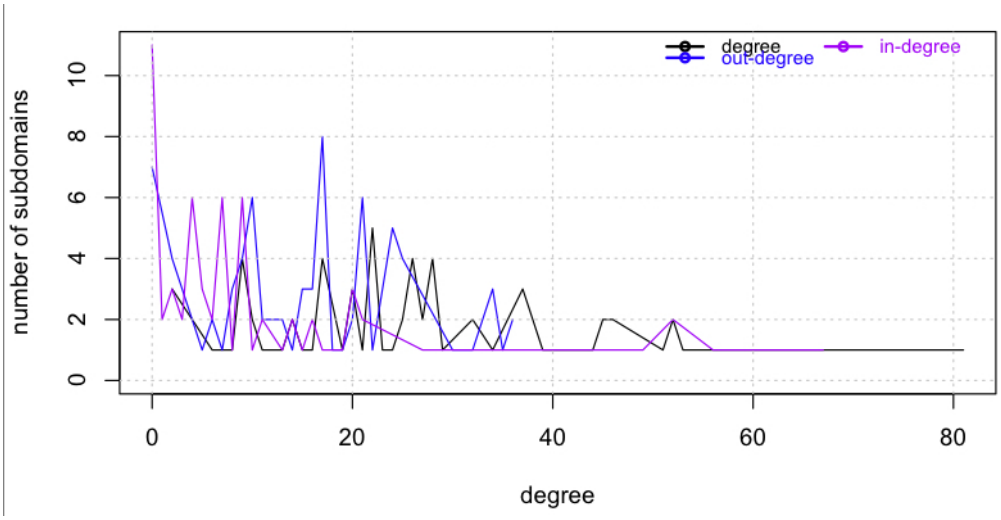

Figure 6.1: Centrality measures for subdomains

We make several observations:

- As mentioned before, a large number of applications (more than 60) are not captured in the application dependency graph as they have a degree of zero. This could have many reasons:
  - The applications can be completely isolated from the rest of the ecosystem.
  - The missing applications can communicate with other applications by other technologies than direct PL/1 calls. This could be a data-communication by means of databases or files. The message based communication is also a possibility.

  In a further version of the tool, where other communication technologies would be implemented, this question could be solved. In this thesis there are no further investigations concerning this problem.
- Figure 6.2 shows that most of the applications have a centrality degree somewhere under twenty. Thus, at the application level one can see an ecosystem where the behavior is well distributed between the applications. In a legacy ecosystem one would not expect such a good distribution of the behavior. But during the last ten years the company invested a lot of resources into reengineering the ecosystem and it looks like this work already paid off.
- For programs the outdegree is always somewhat higher than the indegree and over 70% of the programs have a degree smaller than 5. This is a sign that reuse can be improved at the program level.
- Utility programs exist which have indegree of a few hundreds.

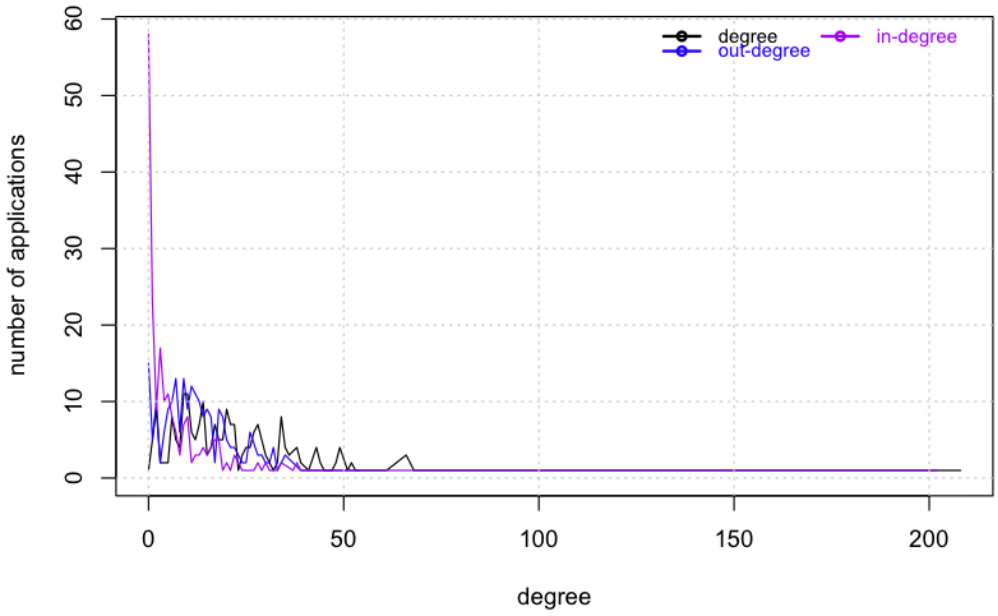

Figure 6.2: Centrality measures for applications

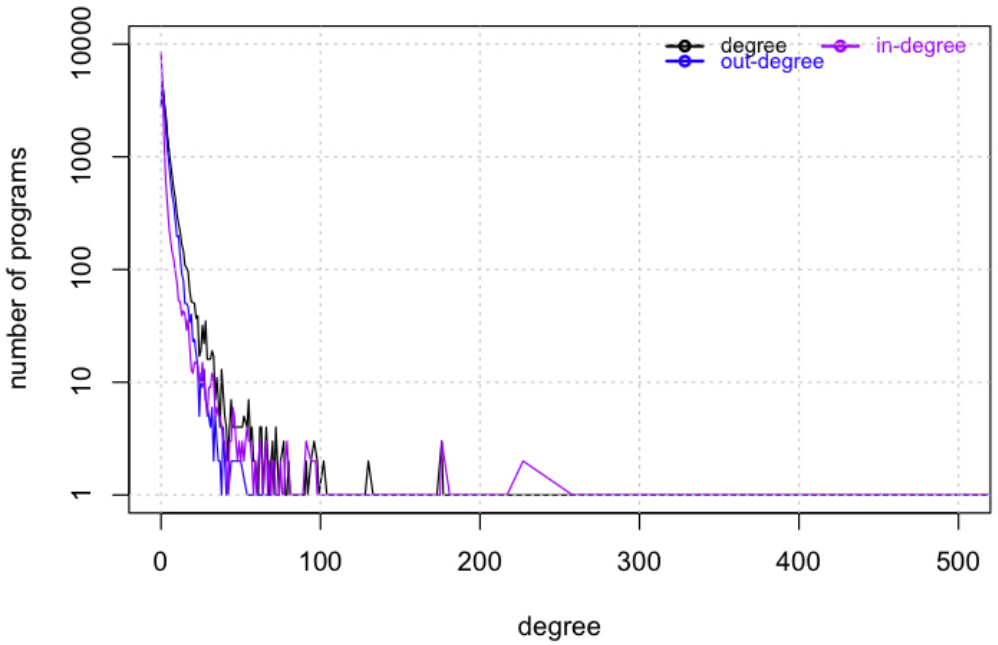

Figure 6.3: Centrality measures for individual programs

7 Discussion

7.1 Results of Discussions with Experts

We organized the second round of interviews after implementing St1-PL/1. This time we also showed a first version of the tool to company employees involved in the design of the development process to find out whether they can gain new insights into the system by using the tool.

We learned that in our tool the stakeholders see benefits in starting with a global view of the domains and the possibility of zooming in to show parts of the ecosystem at the subdomain and application level.

We observed that allowing the respondents to see the tool in action lead them to have new insights into what is possible and what is desirable, and we learned about new requirements which were not present in the first round of interviews:

R4. **Automatically detecting components that need redesign**. People involved in the definition of the development processes are interested in learning which components of the ecosystem must be redesigned. An approach which would automatically pinpoint such components would be valuable.

The work on quality and metrics of Lanza and Marinescu [11] is relevant to this requirement. Particularly, it would be interesting to adapt the detection strategies of Marinescu [15] to legacy PL/1 code.

R5. **Runtime analysis and visualization**. The stakeholders would value information about the runtime behavior of the analyzed systems and the connections between the different technologies as a complement to the static view that the St1-PL/1 infrastructure presents.

There is a rich portfolio of work on which such tools can draw on in visualizing multiple systems running in parallel including the work on Zinsight of De Pauw [3].

R6. **Multiple dependency types**. There was a strong consensus between the stakeholders that we discussed with that an overview tool should integrate multiple types of dependencies, not just calls. Examples are join dependencies on the same database table, CORBA, message queues, files, *etc.*

To gather more requirements, the tool should be shown to a larger group of stakeholders. When working a short time with it they would surely come up with much more concrete and focused requirements.

These new requirements remain to be addressed in future versions of St1-PL/1.

7.2 Lessons Learned

Some of the challenges and the lessons learned in the course of this project are:

- By leveraging existing tools we could move quite fast. We were able to build our analysis infrastructure without much effort by customizing existing frameworks, especially the parsing infrastructure of PetitParser and visualization of Quicksilver.
- Legacy code runs on large systems in large organizations. Figuring out how to get the source code to analyze is the first challenge.
- One must reuse information that is hiding in other tools and other corners of the organization.
- When building tools for legacy ecosystems one must be aware of obsolete practices. The existing automatic code review tools can not enforce the absence of GOTO since the legacy code has it, even if nowadays this is a discouraged practice.
- When working with large amounts of data, performance is an issue. Due to some of the limitations of the infrastructure we relied on and the fact that we could not choose the machine on which to run the analysis, the parsing part of the analysis took many hours.
- While we were working on the project, the company bought an analysis tool – a cross-referencer for programmers. Although the PL/1 software has been developed with the same tools for many years, the practice of software development starts to slowly catch up with the research.
- People are not easily convinced to use a new tool unless they see immediate results and benefits.
- One does not need to implement a complete parser to get a big picture of the source code in a legacy ecosystem.
- One must learn how to do interviews and prepare well before doing them.

7.3 Conclusions

7.3.1 Summary

- This thesis presents an experience report on analyzing a large PL/1 legacy ecosystem hosted in an industrial setting. The analysis shows that by applying the tool on the case study ecosystem it can be successfully used for analyzing large PL/1 ecosystems.
- The thesis also lists requirements for legacy ecosystem analysis tools which were elicited by running interviews with stakeholders in the industrial setting in which the case study ecosystem functions.
- The solution part presents a tool named St1-PL/1 which implements some of these requirements by providing a top-down visual exploratory approach on the ecosystem.
- However, stakeholders, who looked at the tool, added new requirements which would potentially increase its and other similar tools' usefulness. Implementing these requirements remains as future work.

7.3.2 Contribution

The contributions of this thesis are:

- A new kind of a graphical interface that does not exist in the company. Especially the top-level-approach is not very popular within the world of legacy ecosystems running on mainframes.
- By executing some graph-theoretic analysis we were able to measure the legacy ecosystem in a way never done before.
- With the interviews we gathered the needs of the employees working with the code. These people have never been asked before about how they would see a new analyzing tool.

7.3.3 Future Work

The following points could be the subject of further work to improve the tool and extend the significance of the analysis done on the ecosystem.

- Considering the whole code basis and not only the files before precompiling.
- Adding additional metrics to the tool.
- Adding more kinds of information-flow between entities like MQ, Web Service, files or databases.
- Analyzing the ecosystem during run-time in order to get quality metrics and entity connections that cannot be gathered by the statically approach.
- Make a more detailed survey to get a better response to the work we have done and to identify more needs by the employees.
- Working on the performance issues. For solving this the tool should somehow run on a server, but as it is hard to get the source code out of the banking system, the goal would be to find an internal server where our tool could run on.
- Improving the usability and interactivity

Appendices

A.1 Tutorial ST1-PL/1

This section explains in a short way how to install the tool.

A.1.1 Installation

To use the analysis tool following installations are essential:

1. Instalation
   (a) As a first step, an actual image with Softwarenaut has to be installed:
      - source location: http://ci.moosetechnology.org/job/softwarenaut-latest-dev/
      - files: sofwarenaut.changes & softwarenaut.image

   (b) The actual version of the PL1 Tool has to be loaded:
      - source location: http://www.squeaksource.com/PL1Analysis.html
      - file: PL1-ErikAeschlimann.XX.mcz (latest)

   (c) To see the visualization with Grid Layout, the following change has to be done in method *SnVisualizer >>visualRepresentationOF: on:*
      - Replace *ROCircleLayout new applyOn: visualNodes* with *ROGridLayout new applyOn: visualNodes*

2. Files needed
   - Unzip the folder CSCode.zip and put it into the Resources folder of the Moose app. It contains the needed folders and files to run the tool
   - For parsing following folders with the source files are needed in the CSCode folder

A.1.2 Running the tool

Because the parsing of the thousands of files needs more memory than Moose can provide, the run of the tool is separated into two parts:

1. In the first part every domain is parsed separately. So this step has to be repeated for every domain in the ecosystem. The result of this first part are files with calls and LOC's listed. The method to run a parse run is: PL1ParserExample – testParseOneDomainAndWriteAssociations
2. The second part is again split into two subparts:
   (a) First the PL1 Ecosystem with all its entities has to be created and the parse results from the first part have to be loaded: PL1ParserExample — testCreateEcosystemAndLoadAssociations
   (b) Secondly the Glamour visualization has to be started: PL1ParserExample – testGlammour

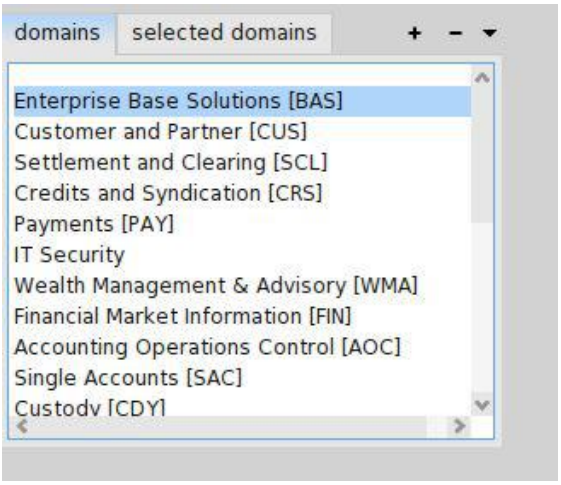

Figure 1: Choosing domain

A.1.3 Use the tool

1. run the tool
   - PL1ParserExample– testGlammour
2. use UI
   - choose domains
     To choose a domain, mark it by clicking on it in the domains tab and add this domain by clicking on the '+' button to the selected domains list. Repeat this for all domains you want to visualized. With the '-' button a marked domain can be removed from the selected domains list
   - change to the selected domains tab
   - when clicking the refresh button the subdomains and applications of the selected domains appear in separate lists
   - choose visualisation of ecosystem:
     - When clicking on the open button in the selected domains tab, the selected domains are displayed on domain level with the relations between them. See Figure 2
     - When clicking on a subdomain in the sudomains list all subdomains of the selected domains are displayed on level subdomains with relations between them. The relations from and to the chosen subdomain are marked with a red color. See Figure 3
     - When clicking onto an application in the applications list, then all applications of the selected domains are displayed on application level with the relations between them. The relations from and to the chosen application are marked with a red colour. See Figure 4

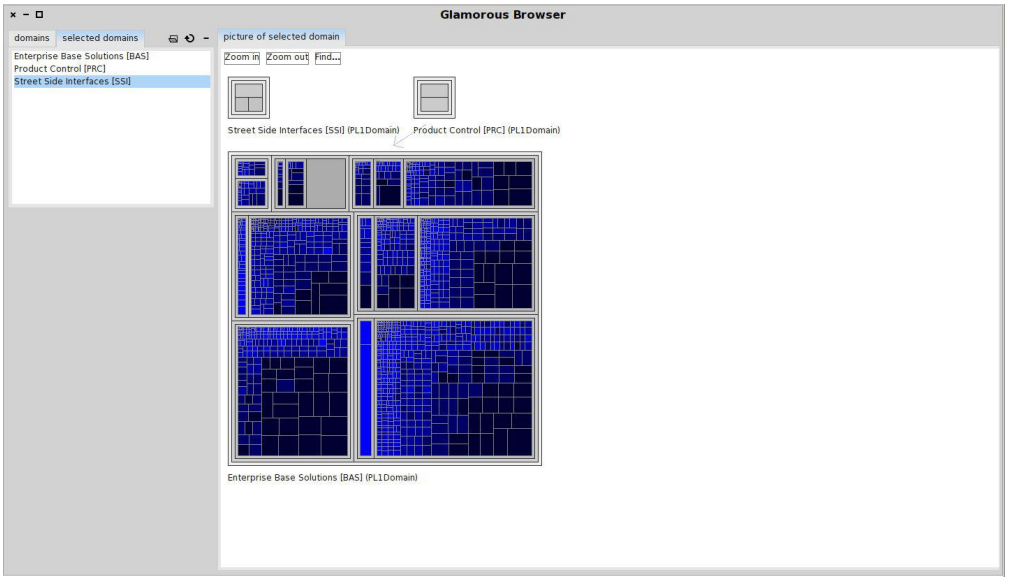

Figure 2: visualizing selection

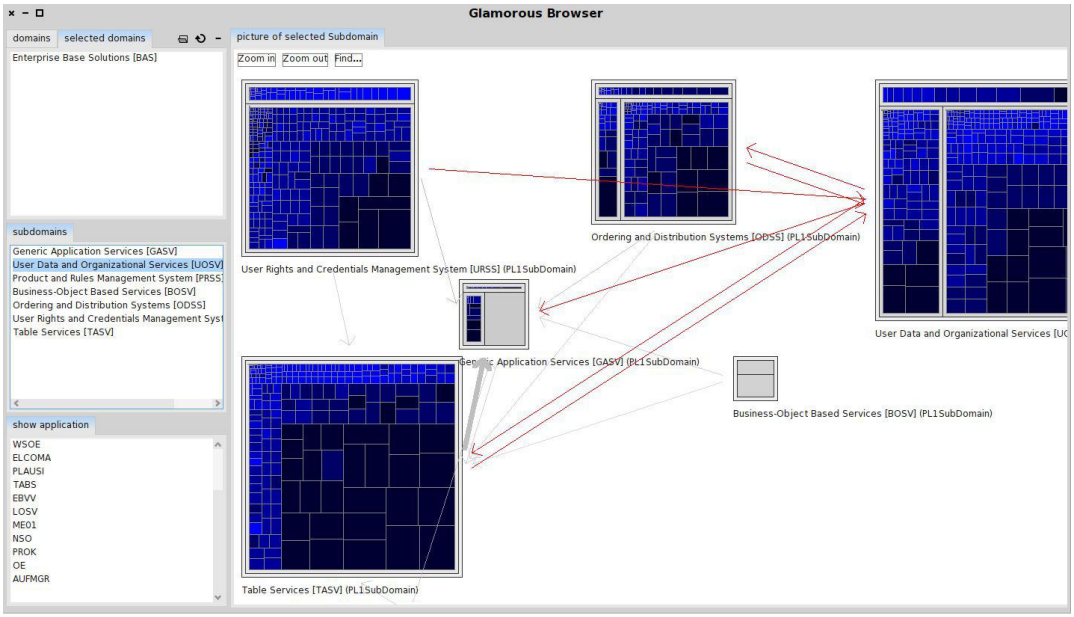

Figure 3: Visualizing on subdomain level

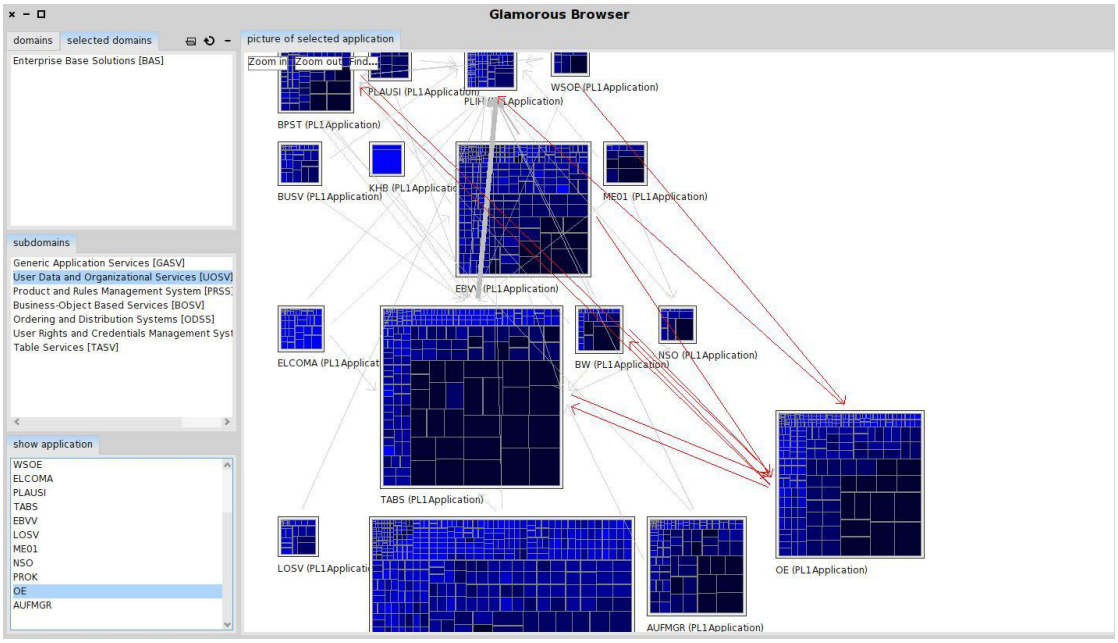

Figure 4: Visualizing on application level

Bibliography

[1] Bas Cornelissen, Danny Holten, Andy Zaidman, Leon Moonen, Jarke J. van Wijk, and Arie van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *Proceedings of the 15th International Conference on Program Comprehension (ICPC)*, pages 49–58. IEEE Computer Society, 2007.

[2] A. De Lucia, G.A. Di Lucca, A.R. Fasolino, P. Guerra, and S. Petruzzelli. Migrating legacy systems towards object-oriented platforms. In *Software Maintenance, 1997. Proceedings., International Conference on*, pages 122–129, 1997.

[3] Wim De Pauw and Steve Heisig. Zinsight: a visual and analytic environment for exploring large event traces. In *Proceedings of the 5th international symposium on Software visualization*, SOFTVIS '10, pages 143–152, New York, NY, USA, 2010. ACM.

[4] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[5] Edsger W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.

[6] Jürgen Ebert, Bernt Kullbach, Volker Riediger, and Andreas Winter. Gupro - generic understanding of programs. *Electronic Notes Theoretical Computer Science.*, 72(2), 2002.

[7] Nicole Haenni, Mircea Lungu, Niko Schwarz, and Oscar Nierstrasz. Categorizing developer information needs in software ecosystems. In *Proceedings of the 1st Workshop on Ecosystem Architectures*, pages 1–5, 2013.

[8] Brian Johnson and Ben Shneiderman. Tree-maps: a space-filling approach to the visualization of hierarchical information structures. In *VIS '91: Proceedings of the 2nd conference on Visualization '91*, pages 284–291, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[9] B. Kullbach, A. Winter, P. Dahm, and J. Ebert. Program comprehension in multi-language systems. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'98)*, WCRE '98, pages 135–, Washington, DC, USA, 1998. IEEE Computer Society.

[10] Michele Lanza. CodeCrawler — a lightweight software visualization tool. In *Proceedings of VisSoft 2003 (2nd International Workshop on Visualizing Software for Understanding and Analysis)*, pages 51–52. IEEE CS Press, 2003.

[11] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[12] Mircea Lungu. *Reverse Engineering Software Ecosystems*. PhD thesis, University of Lugano, November 2009.

[13] Mircea Lungu, Michele Lanza, Tudor Gîrba, and Romain Robbes. The Small Project Observatory: Visualizing software ecosystems. *Science of Computer Programming, Elsevier*, 75(4):264–275, April 2010.

[14] Mircea Lungu, Romain Robbes, and Michele Lanza. Recovering inter-project dependencies in software ecosystems. In *ASE'10: Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*. ACM Press, 2010.

[15] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 350–359, Los Alamitos CA, 2004. IEEE Computer Society Press.

[16] Leon Moonen. Generating robust parsers using island grammars. In Elizabeth Burd, Peter Aiken, and Rainer Koschke, editors, *Proceedings Eight Working Conference on Reverse Engineering (WCRE 2001)*, pages 13–22. IEEE Computer Society, October 2001.

[17] H. A. Müller and K. Klashinsky. Rigi — a system for programming-in-the-large. In *ICSE '88: Proceedings of the 10th international conference on Software engineering*, pages 80–86. IEEE Computer Society Press, 1988.

[18] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York, NY, USA, September 2005. ACM Press. Invited paper.

[19] J. Ossher, S. Bajracharya, and C. Lopes. Automated dependency resolution for open source software. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 130 –140, may 2010.

[20] Fabrizio Perin. *Reverse Engineering Heterogeneous Applications*. Phd thesis, University of Bern, November 2012.

[21] Fabrizio Perin, Lukas Renggli, and Jorge Ressia. Ranking software artifacts. In *4th Workshop on FAMIX and Moose in Reengineering (FAMOOSr 2010)*, 2010.

[22] Lukas Renggli, Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Practical dynamic grammars for dynamic languages. In *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*, Malaga, Spain, June 2010.

[23] Romain Robbes and Mircea Lungu. A study of ripple effects in software ecosystems (nier). In *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*, pages 904–907, May 2011.

[24] Dennis Schenk and Mircea Lungu. Geo-locating the knowledge transfer in stackoverflow. In *Proceedings of the 5th International Workshop on Social Software Engineering*, pages 21–24, 2013.

[25] Margaret-Anne Storey, Casey Best, and Jeff Michaud. SHriMP Views: An interactive and customizable environment for software exploration. In *Proceedings of International Workshop on Program Comprehension (IWPC '2001)*, 2001.

[26] Sander Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Bern, December 2001.

[27] A.J. Tyrrell. *Eiffel Object-Oriented Programming*. MacMillan Press, 1995.

[28] Richard Wettel and Michele Lanza. CodeCity. In *Proceedings of WASDeTT 2008 (1st International Workshop on Advanced Software Development Tools and Techniques)*, 2008.

[29] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 531–540, New York, NY, USA, 2008. ACM.