Analyzing PL/1 Legacy Ecosystems: An Experience Report

Erik Aeschlimann, Mircea Lungu, Oscar Nierstrasz
University of Bern
Switzerland
http://scg.unibe.ch/

Carl Worms
Credit Suisse, AG
Switzerland
carl.f.worms@credit-suisse.ch

***Abstract*—This paper presents a case study of analyzing a legacy PL/1 ecosystem that has grown for 40 years to support the business needs of a large banking company. In order 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 and then present results of analysing the case study. We show that although the vision for the future is to have an ecosystem architecture in which systems are as decoupled as possible the current state of the ecosystem is still removed from this. We also present some of the lessons learned during our experience discussions with stakeholders which include their interests in automatically assessing the quality of the legacy code.**

I. Introduction

Software systems do not exist in isolation but are rather parts of larger ecosystems in which multiple software systems interact and co-evolve together. The development and maintenance of these ecosystems entails new challenges that are partly due to scale. Nevertheless, some of the static analysis techniques that have been used at the individual system level can also be applied on entire ecosystems [13].

Large commercial software ecosystems have rarely been empirically studied but instead researchers have recently focused on the analysis of open-source software: from the empirical study of their evolution patterns [25], to categorizing the information needs of software developers working in such open-source ecosystems [7], [8].

In this paper we approach a different type of ecosystem: a large, multi-language, enterprise system that grew over many decades. The system is composed of hundreds of applications that collaborate and co-evolve in parallel while being developed by multiple teams working in distributed contexts. We apply traditional reverse engineering techniques on its source code, and report on the results. However, to keep the work manageable we only analyze those systems that are written in PL/1. The contributions of this paper are:

1) Presenting requirements for analysis tools aimed at large-scale industrial legacy ecosystems
2) Introducing a tool that supports the analysis and visualization of PL/1 ecosystems
3) Analyzing a real-world large-scale legacy ecosystem and providing insight into its magnitude, complexity, and associated problems

II. A Brief Overview of PL/1

PL/1 is a third-generation procedural programming language that was created in the 1970's by combining 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 1 shows a typical PL/1 fragment. 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.

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
```

Fig. 1. PL/1 program

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

- *Procedures.* Line 001 of Figure 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

978-1-4799-2931-3/13 © 2013 IEEE

WCRE 2013, Koblenz, Germany
Practice Track

program. Include files contain code fragments that are inserted into the program files during a preprocessing step.

- *Comments.* Line 003 shows a PL/1 comment. In addition to the documentation of the intent of a program, comments are especially important to communicate information that cannot easily be read in the code, like external programs, used files or used databases.
- *Variable Declarations.* Line 5 shows a variable declaration. 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.
- *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.
- *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].
- *Calls.* Line 017 illustrates the call statement used in PL/1 to invoke external programs.

The PL/1 compiler underwent a major revision seven years ago with no backward compatibility, so the company had to migrate the complete PL/1 codebase to the new compiler version with the help of offshore companies. This migration was also used as an opportunity for some redesign, like replacing assembler code with standard functions for date/time conversion as well as some minor code redesign (*e.g.*, goto replacement)

III. A PL/1 Ecosystem

We have been granted access to a complex legacy ecosystem consisting of hundreds of applications written in several programming languages and multiple technologies. It had evolved for over 40 years to support the back-end business processes of a large banking company.

In this work we focus our attention on the PL/1 part although approaches to program understanding for multi-language systems have been proposed before [10], [22]. The hundreds of PL/1 applications interact with each other via services or static dependencies. They have a certain degree of independence in their evolution [13], [17] and are composed of multiple programs.

In total they consist of 30 MLOC before preprocessing and 130 MLOC after. We perform static analysis of these systems and we limit our analysis to the source code before preprocessing. This limitation does not change the soundness of the analysis because the metrics discussed in this paper are not strongly affected by the preprocessor statements.

There are only a few PL/1 applications that run in isolation 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 services.

Due to the fact that there is an ample number of applications and that only recent versions of the PL/1 compiler support the 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 the subdomains contains multiple applications that logically belong together.

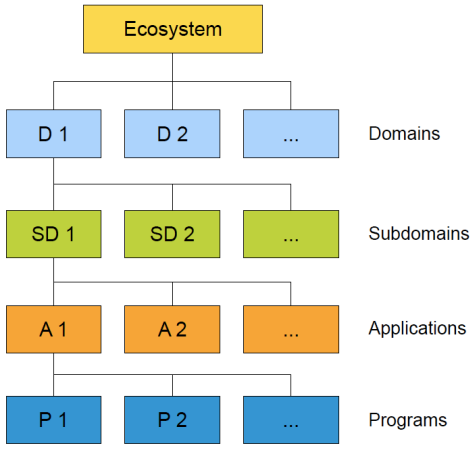

Fig. 2. The applications in the ecosystem are organized in a hierarchy with two levels

Figure 2 illustrates the four levels into which the source code is structured. At the top level there are about 30 *domains*. The domains and subdomains emulate a package system that has only recently been incorporated into PL/1.

Evolving such a large legacy code base involves particular challenges that we expected would be supported by specific tools for monitoring its evolution. However, we learned that there were not many tools that provide the big picture to support the evolution process.

IV. Ecosystem Analysis Requirements

We conducted several free form interviews with employees aiming to identify their needs and wishes concerning tool support for analyzing the PL/1 ecosystem.

To conduct the interviews the first author of this paper interviewed people covering 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.

The interviews led us to several requirements for analysis tools:

R1. **Produce high-level views of the entire code base.** Tool support for presenting a 50,000 foot view of the entire

code-base is missing. All of the interviewed employees use standard mainframe tools to analyse the code, like cross-reference tools, performance measuring tools and dump analysis tools. None of these tools features graphically visualized output showing the big picture of the ecosystem.

The interviewees provided various reasons why 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 need 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 application landscape, strong, undesired, coupling relationships evolved between the domains and subdomains during the evolution of the ecosystem.

A long-term goal of the company is to decouple the individual systems and sub-systems and increase the independence and potential of the different applications to evolve independently. This would simplify the replacement of whole subsystems with alternatives written in more popular languages or third party products.

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

On the other hand, we observe that these requirements are quite specific to industrial legacy systems as we have seen in another recent study which shows that developers in open source systems have different information needs, which steams from different motivations such the desire to keep up with evolving upstream and downstream systems [7].

V. The ST1-PL/1 Infrastructure

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 [20].

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

In the following we briefly present the individual components and discuss some particular choices we made to adapt the infrastructure to our case study. We discuss in turn the following three components:

1) A model extractor based on island grammars [18] is built with the PetitParser framework [24].
2) A meta-model that is an extension of FAMIX is populated with information extracted from the source code and other external sources of information.
3) Graph analysis and visualization tools are built to work on top of this meta-model.

A. The Model Extractor

To extract the information required by the meta-model we had to parse the source code as well as external sources of information like the ones mentioned in section III. For the code we used PetitParser, a parser framework based on PEGs (parsing expression grammars) [24]. 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 [18].

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]

Fig. 3. An external file contains the mapping between the programs and their external organization

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 3 shows a fragment of the extracted hierarchy data.

B. The Meta-model

In order to leverage the Moose analysis platform and the tools it offers, we extended FAMIX [29] (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 the applications in our case study.

Figure 4 illustrates the extended meta-model. The elements in the figure are:

- FAMIX classes, shown in white.
- PL/1 specific classes, in blue.
- Attributes added to standard FAMIX entities are shown in grey.
- Ecosystem-specific entities indicating structural levels are yellow.

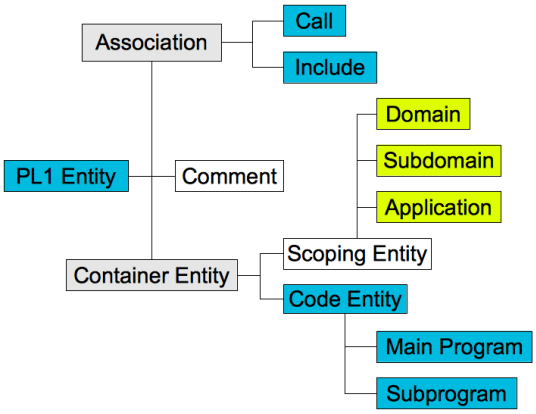

Fig. 4. Meta-model of PL/1 Ecosystem

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

C. The Analysis

St1-PL/1 supports several types of analysis: metrics, interactive visualization, and graph analysis.

We compute metrics that 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, FAN-OUT — sums up all incoming and respectively outgoing calls to an entity.
3) 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.
4) FP — function points. 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 contained entities.

The visualizations are prepared with Quicksilver[2], a port of our previous Softwarenaut [15] visualization tool to Pharo Smalltalk[3]. The tool is interactive and supports an "overview, zoom, details on demand"[26] exploration approach in the tradition of classical reverse engineering tools such as Rigi and Shrimp [19], [28].

VI. The Analysis

In this section we report on the analysis performed using St1-PL/1. We do not discuss the parsing and model building parts but instead focus on the analysis. We start by presenting a top-down exploration session and continue with providing a series of network theoretical measurements on the studied ecosystem.

[1]http://www.emn.fr/z-info/atlanmod/index.php/Ecore
[2]http://scg.unibe.ch/research/quicksilver
[3]http://pharo-project.org

A. Top-Down Exploration

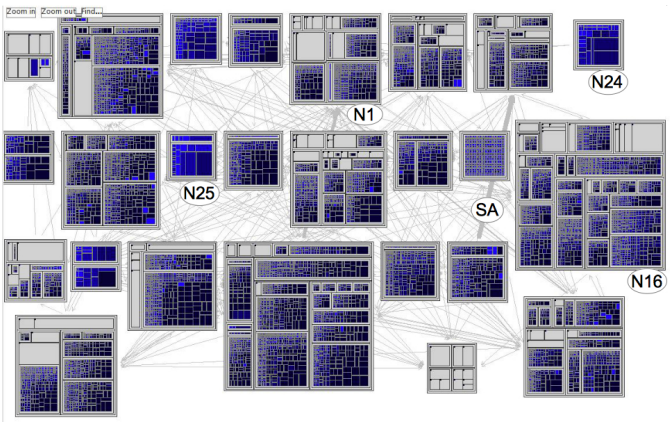

Fig. 5. Visualization of the whole PL/1 ecosystem showing call relationships between domains and highlighting function points

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 presents the overview of the domains and their relationships. Here we detail several 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 [9] 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. Since the human eye cannot distinguish between too many shades of blue, we map all the values to five distinct shades that 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.

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 generally 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.[4]

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?

Interaction in St1-PL/1

Given the importance of domain N1 we decide to zoom in and learn about the quality of the code inside it (R2). As one of the aspects of quality we check the compliance with coding style guidelines, particularly the usage of GOTO instructions which the guidelines strongly discourage.

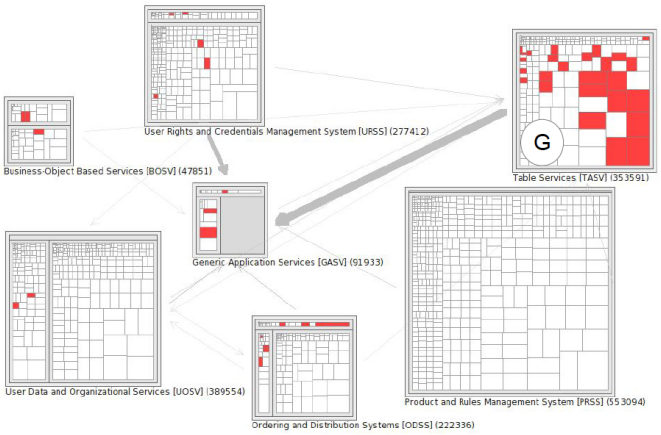

Fig. 6. Visualizing the usage of GOTO inside the domain N1

Figure 6 shows the view that we obtain by interactively zooming into N1 with St1-PL/1. In the figure, every program that contains GOTO statements is highlighted in red. One can see that the domain marked (G) in the top-right corner is a candidate for further inspection.

B. Network Analysis

To better understand the nature and complexity of our case study we complement the description of our interactive analysis with a network theoretical analysis of the case study facilitated by St1-PL/1. Network analysis has been applied to software for various purposes including predicting modules that are prone to have defects in the future [31]. The reported results will be of interest and a reference point to researchers and managers of large legacy ecosystems.

[4]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.

Graph-Level Indices

The domain, subdomain, application, and program graphs are directed graphs, so we can analyze them as such. We start by computing the density of each of the graphs.

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.

TABLE I
INDICES GAINED ON GRAPH LEVEL (GLI'S)

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

Table I 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%. Weak coupling between the applications is good, but it could also mean that reusable code is not centralized enough into infrastructure applications.

Node-Level Indices

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 [23].

We compute the centrality of the nodes at the different abstraction levels in the ecosytem with four measures: degree centrality, indegree, outdegree, and PageRank. The degree of a node is defined by the sum of all incoming and outgoing connections. Indegree only counts incoming and outdegree only counts outgoing connections of the node. PageRank weights the incoming links based on the importance of their source.

TABLE II
INDICES GAINED ON LEVEL DOMAIN

	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 II 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 and discussed earlier)

groups infrastructure applications that provide basic functions to numerous other systems in all the domains. Several other infrastructure domains have zero outdegree. However, there is no domain that dominates the others based on PageRank – the functionality is balanced.

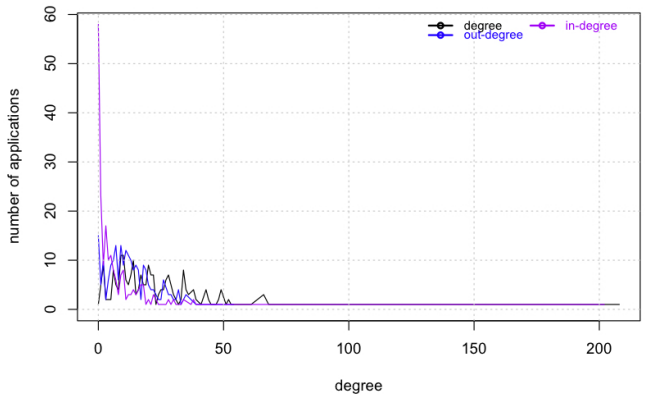

Fig. 7. Centrality measures for applications

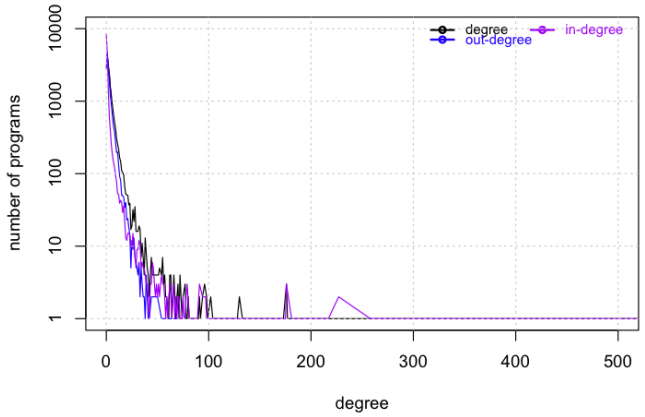

Fig. 8. Centrality measures for individual programs

Figure 7 and Figure 8 show the distribution of centrality in the ecosystem at the application and program level. We see that as the number of vertices in the graph increases (as listed in Table I) the degree spectrum also increases.

We make several observations:

- A large number of applications (more than 60) are not captured in the application dependency graph as they have a degree of zero. We do not know whether they are completely isolated from the rest of the ecosystem, or if they are interacting through other means (*e.g.*, services, database, etc.).
- 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.
- 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 that have an indegree of several hundreds.

VII. Discussion

A. Second round of interviews

We organized the second round of interviews after implementing St1-PL/1. This time we also showed the tool to company employees involved in the design of the development process to find out what they think about it. From these second interviews we learned that 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 that were not present in the first round of interviews:

R4. **Automatically detecting parts of the ecosystems that need redesign**. The stakeholders defining development processes are interested in learning where to focus the redesign processes. An approach that would automaticallypinpoint candidate applications or domains would be valuable. It could draw inspiration from the work on quality and metrics of Lanza and Marinescu [12] by combining multiple system metrics into higher-level detection strategies.

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 ST1PL/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 amongst the stakeholders 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.*

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

B. Lessons learned

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

- By leveraging existing tools we could move quite quickly. 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.
- When building tools for legacy ecosystems one must be aware of obsolete practices. The existing automatic code review tools cannot enforce the absence of GOTO since

it is part of the legacy code, 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.

C. State of the Tools in Industry

The company that provided us with the case study uses IBM-specific tools for their PL/1 development. XREF for example provides a list of programs, that are affected by changes to a subprogram (an external subroutine). This tool does not deliver complete information about the organization of the listed programs. Recently, Panorama[5], a third-party application that computes many metrics of the ecosystem and also creates some visualizations has been used in the company. It computes metrics of the ecosystem and creates visualizations. Its main usage is in searching the code base with a short response time.

Neither Panorama nor the other tools offer the top-down approach we are working on, and we envision our St1-PL/1 to be used as a complement to it. The operating system on the mainframe provides some searching tools that 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.

VIII. Related Work

The work presented in this paper is related to reverse engineering, large scale software analysis and visualization.

The general context in which St1-PL/1 is used is re-engineering. 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.* showing the migration of legacy systems towards OO systems [2].

The idea of automatically aggregating dependencies from the lower level artefacts was first used by Muller in Rigi [19]. 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 used in other tools such as Shrimp [27].

By combining polymetric views [11] with treemaps our visualization can convey the same type of information as CodeCity [30]. However, the advantage of our visualization is that it also 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.

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

Ossher *et al.* resolve dependencies between projects in order to obtain a successful build of a target project in a Java repository of systems [21]. In our previous work we analyze and detect static relationships between the systems in a Smalltalk ecosystem [16].

Also we have developed a tool for visualizing the evolution of object oriented software ecosystems and project repositories that we dubbed the Small Project Observatory [14]. The infrastructure in this paper is targeted towards procedural languages instead, and does not leverage evolutionary information that was not available.

IX. Conclusions

In this paper we have presented an experience report on analyzing a large PL/1 legacy ecosystem hosted in an industrial setting. We have presented 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. We have then presented a tool named St1-PL/1 which implements some of these requirements by providing a top-down visual exploratory approach on the ecosystem. By applying the tool on the case study ecosystem we have illustrated that it can be successfully used for analyzing large PL/1 ecosystems. However, after showing the tool to stakeholders we learned of new requirements that would potentially increase its and other similar tools' usefulness. Implementing these requirements remains as future work as well as taking into account the preprocessor statements for the static analysis.

X. Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Assessment" (SNSF project No. 200020-144126/1, Jan 1, 2013 - Dec. 30, 2015). We thank our colleagues from Credit Suisse IT in Switzerland who contributed to this work through interviews and through their expertise on IT Panorama. We also thank CHOOSE, the special interest group for Object-Oriented Systems and Environments of the Swiss Informatics Society, for its financial contribution to the presentation of this paper.

References

[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] Slinger Jansen. How quality attributes of platform architectures influence software ecosystems. In *Proceedings of the 1st Workshop on Ecosystem Architectures*, 2013. To Appear.

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

[15] Mircea Lungu, Michele Lanza, and Oscar Nierstrasz. Evolutionary and collaborative software architecture recovery with Softwarenaut. *Science of Computer Programming (SCP)*, 2012.

[16] 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.

[17] David G. Messerschmitt and Clemens Szyperski. *Software Ecosystem: Understanding an Indispensable Technology and Industry*. The MIT Press, 2005.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

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

[24] 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.

[25] Romain Robbes, Mircea Lungu, and David Roethlisberger. How do developers react to API deprecation? The case of a Smalltalk ecosystem. In *Proceedings of the 20th International Symposium on the Foundations of Software Engineering (FSE'12)*, pages 56:1 – 56:11, 2012.

[26] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *IEEE Visual Languages*, pages 336–343, College Park, Maryland 20742, U.S.A., 1996.

[27] 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.

[28] C. Stork, V. Haldar, and M. Franz. Generic adaptive syntax-directed compression for mobile code, 2000.

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

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

[31] 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.