Traceability between Function Point and Source Code

Paulo José Azevedo Vianna Ferreira
Universidade Federal do Estado do Rio de Janeiro
Av. Pasteur, 458 – Urca, Rio de Janeiro, RJ, Brasil
55 21 2530-8051 / 3873-4015
paulojose.ferreira@uniriotec.br

Márcio de Oliveira Barros
Universidade Federal do Estado do Rio de Janeiro
Av. Pasteur, 458 – Urca, Rio de Janeiro, RJ, Brasil
55 21 2530-8051 / 3873-4015
marcio.barros@uniriotec.br

ABSTRACT
Software development can achieve interesting benefits through the use of requirements traceability, including improved program comprehension, easier maintenance, component reuse, impact analysis, and measure of project progress and completeness. On the other hand, while the cost of a new IS can be estimated by applying Function Point Analysis, this technique has limited application on maintenance. By determining the impact of changing a given set of features, IS development organizations can build a clear understanding of the effort that these changes will require. In this paper, we propose a technique which uses traceability to build a bridge between function points and source code. We believe that this technique can support negotiations between IS development organizations and their clients regarding changes to Information Systems.

Categories and Subject Descriptors
D.2.1 **[Requirements/Specifications],**
D.2.7 **[Distribution, Maintenance, and Enhancement].**

General Terms
Documentation, Design, Experimentation.

Keywords
Requirements Traceability, Function Point, Source Code.

1. INTRODUCTION
Requirements Traceability is the "ability to describe and follow the life of a requirement, in both a forward and backward direction" [1]. By using traceability links, the stakeholders rationale can be followed from requirement documents to design models, source code, test cases, and virtually any artifact built as part of a software project.

Benefits which can be achieved through traceability include better program comprehension, easier maintenance, identification of reusable components, change impact analysis and an evaluation of project progress and completeness [2, 3]. But despite these benefits, current techniques and tools are not mature enough to maintain sound and complete traceability links in a fully automated manner [3], thus requiring manual intervention. Moreover, due to deadline pressures, developers tend to set aside activities not directly related to delivering the software, and traceability is common among these supporting tasks. Therefore, traceability is not as widely adopted in software industry as it could, according to its potential benefits.

Information systems (IS) are not an exception to this scenario. While some IS development companies complain of their inability to manage a changing set of user requirements, they cannot accurately state which artifacts are responsible for implementing each requirement. Thus, they cannot provide the client with clear arguments on the cost of developing a new feature, and the impact it may have upon existing features.

Function Point Analysis (FPA) [4] has become widely used to measure system complexity (size). In FPA, the functionality provided by a system is modeled as *data functions* and *transactional functions*. Data functions represent the complexity of the data handled by the IS, while transactional functions represent the complexity embedded in the processes by which such data is gathered, handled, and transformed. The information required to elicit data and transactional functions can be collected from users early in the development life cycle, and this is commonly used as a coarse description for system requirements, in order to support development contracts and service pricing.

Since pricing and contract issues in IS development are usually related to function point models (data and transactional functions), traceability links between these models and software artifacts that take part in their implementation can be useful for supporting discussions on the impact of a functionality to be introduced into the system, as well as supporting the maintenance of a system from an early representation of its requirements.

This paper presents a technique to generate trace links between function point models and source code elements in a two-step approach: one to relate data functions to source code and the other to create relationships between transactional functions and the source code. The proposed approach requires relatively low effort on gathering information in that it is semi-automatic, avoiding a large increase on the total development effort. We have conducted three case studies to address the feasibility of generating trace links from FP data and transactional functions, and produced a prototype to demonstrate how the development team could access trace links in its own daily-work tool.

This paper is organized in eight sections: the first includes this introduction; the second is a brief presentation of FPA; the third details our approach; the fourth presents the case studies; the fifth presents the prototype; the sixth presents threats to validity; the seventh presents some related work; we finish in the eighth section by presenting some conclusions, contributions, limitations and future work.

TEFSE'11, May 23, 2011, Waikiki, Honolulu, HI, USA

2. FUNCTION POINT ANALYSIS

One of the first papers about Function Point Analysis (FPA) referred to its use in productivity measurement and effort estimation [4]. Over time, it became a widely adopted method to estimate effort, time, cost, and resources required to conduct an IS development project, as well as supporting negotiations between IS development companies and their clients.

FPA estimation is divided into two main phases: calculating an unadjusted number of function points (UFPs) and applying 14 predefined adjustment factors to calculate the number of adjusted function points (AFPs). The effort required to build the system can be derived from the latter. However, to calculate the number of UFPs, an analyst must address the system in two perspectives: the data it maintains (data functions) and the transactions that handle these data (transactional functions). Each data and transactional function yields a number of UFPs, which are summed to achieve the total number of UFPs.

Data functions are based on domain concepts, identified and organized into logically related data groups maintained or queried by the system. These data groups are further classified as Internal Logical Files (ILFs), if they are maintained by the system (the system feeds data into the group or changes the data it maintains), or External Interface Files (EIFs), if they are maintained by another system and queried by the current one. Each data group comprises recognizable subsets of data (RETs) and atomic, non-repetitive, user-recognizable attributes (DETs). The complexity of each data group is calculated according to its RETs and DETs and represented in a three-valued ordinal scale (low, medium, or high). Finally, the number of UFPs contributed by each data group is defined based on its complexity.

In the FPA context, transactions are elementary processes that handle data as it moves across the boundaries of the system. Processes receiving data from outside the system are classified as External Inputs (EIs). Processes collecting data maintained by the system and sending these data to some receiver outside the system are classified as External Outputs (EOs) or External Inquiries (EQs), depending on whether they apply complex processing upon the information before it crosses the border of the system. If such processing (algebraic formulas, data validation rules, complex grouping or sorting) is required, the transaction is classified as an EO; otherwise, it is classified as EQ.

A transaction is characterized by the number of distinct data groups that it uses (FTRs) and the number of atomic, non-repetitive attributes (DETs) that it collects from or shows to the user. The complexity of each transaction is calculated based on its FTRs and DETs, and represented in a three-valued ordinal scale (low, medium, or high). As with data functions, transaction complexity defines the number of UFPs with which it contributes.

A usual function point model describes system data groups by listing their data subsets and attributes. It also presents the transactions provided by the system, reporting on their referenced data groups and field attributes. The effort required to conduct a software development project is frequently derived from this model and, as the user requires new functionalities, the model is updated and a new total effort is calculated. The effort to execute the required changes is calculated as the difference between the original effort and the updated one. Thus, FPA is useful both as the basis for system requirements and for negotiation between the company and its clients.

3. AN FPA-BASED TRACE LINK GENERATION TECHNIQUE

This section describes our proposed technique to generate trace links between FP data and transaction models and source code modules. Our approach requires the collection of the following information about the IS under interest:

(a) An FP data model, described as the set of ILFs and EIFs used by the system, along with their RETs and their DETs. The current approach does not directly distinguish ILFs and EIFs;

(b) A set of database tables and their fields, is usually collected from a relational database schema. Our approach is limited to IS which store data in relational databases;

(c) A manually produced set of trace links between data functions and database tables, which supports the identification of trace links between data functions and the source code;

(d) An FP transaction model, described as a set of transactions provided by the system (including EIs, EOs, and EQs);

(e) A set of test cases to evaluate every transactional function supported by the system;

(f) A manually produced set of trace links between transactional functions and test cases, which supports identifying trace links between transactional functions and the source code;

(g) The system source code as a directory structure of the files which convey its classes, routines, and scripts. The source code must be configured so that it can be executed under a code coverage tool. Thus, an operational environment must be set to execute the system, including its database and other resources referenced by the source code;

(h) A selected *granularity* [5] by which some modules of the source code will be mapped to data and transactional functions (e.g. packages, files, classes or lines of code).

Semi-automated traceability approaches usually require an initial set of trace links in order to discover additional links with less effort than manual approaches [5]. Our technique uses a different approach, divided into two steps: a pattern-based mapping of the data model to the source code which handles data and an execution-based analysis of the code which supports each transaction. Nevertheless, as other traceability approaches, the results of our technique depend on the quality of the input. Figure 1 presents an overview of our proposed technique.

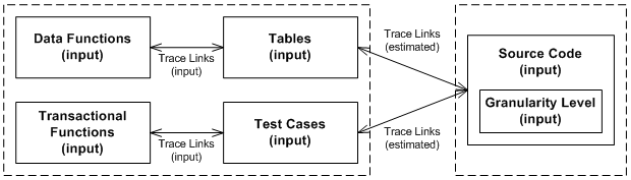

Figure 1. Overview of the technique.

In the first step, the user must map the system's ILFs to tables in the database schema. EIFs are not taken into account, since they represent data stored outside the system database. Afterwards, our technique searches the code for queries to database tables and creates trace links between database tables and source code modules, according to the selected granularity.

In the second step, the user must select a set of scenarios that run each transactional function presented in the transaction model. A code coverage tool monitors the source code modules are executed while a test automation tool executes each usage scenario. By combining this information with a timeline of scenario execution, we can generate links between the transactions and the code modules participating in their implementation.

According to [5], each organization or project team must define its need for traces at a certain level of detail which can range from coarse-grained artifacts (e.g. packages, class diagrams) to fine-grained ones (e.g. individual lines of code, classes, attributes). Tracing fine-grained artifacts can offer better results, but it is more expensive and the collective set of trace links are harder to maintain than traces built for coarse-grained artifacts. Thus, a cost-benefit relation influences this decision.

The case studies presented in section 4 evaluated whether our technique could generate trace links from ILFs to source code modules. However, a few changes in the automatic scripts could enable processing finer-grained levels. For instance, DETs could be traced by searching data fields in queries. An important aspect is the ability of the selected code coverage tool to identify which modules, methods or source code lines were enacted by each scenario. Limitations on the code coverage tool influence the level of detail by which source code can be traced to transactions.

3.1 Data Function Traces

Some differences exist between an FP data model and its implementation as a relational database. Due to implementation and performance restrictions, the system relational database may not directly represent its FP data groups. For instance, FP data models do not represent table relationships and primary keys, required by the relational database. Moreover, the database schema states how data is stored by the system, while the FP data model states which data is required to allow the system to fulfill its transactions.

Thus, the database schema is an implementation of an FP data model. The source code usually references the database schema through queries and data manipulation commands (INSERT, DELETE etc.). Our technique uses the system database schema as a bridge between the FP data model and the source code. An analyst who knows both the domain area and these models can manually trace ILFs to the tables composing the database schema.

References to tables are searched in the source code. Trace links are generated between referenced tables and source code modules where references were found. The more references exist, the stronger the relation represented by the trace link is: the number of references indicates the weight of trace links. This weight represents how much a table is handled by a source code module and inversely how much a source code module is responsible for handling the information stored in that table.

Considering that we have trace links from the FP data model to the database schema and from the database schema to source code modules, we apply the property of transitivity in the context of trace links, leading to what we named *indirect trace links* between the FP data model and the source code. The weight of an indirect trace link is derived by summing weights of all schema-to-code links which belong to a specific indirect trace link.

3.2 Transactional Function Traces

Usage scenarios (i.e. test cases) can be used to generate trace links between these scenarios and source code [6]. A code coverage tool monitors source code modules which are activated while these scenarios are being executed. According to the ability of the code coverage tool to monitor the code in different levels of detail, classes, methods, or even individual lines can be identified.

Our technique applies this testing and monitoring approach in order to trace transactions to source code. A set of scenarios must be defined to test every transaction presented as part of an FPA transaction model. While these scenarios are being executed, a code coverage tool monitors and records the exact moment when those source code modules were activated.

Therefore, our technique uses test cases developed to ascertain the implementation of the system under analysis as a bridge between FP transactional model and the source code. An analyst who knows both the domain area and the test cases can manually trace EIs, EOs and EQs of transactional model to test cases.

We need to record the start and end execution time for each test case. By analyzing code coverage records, we can identify which source code modules were enacted during each test case execution. Trace links are generated between test cases and source code modules activated during the execution of the test cases. As with data functions, the more activated lines of code exist, the stronger the relation represented by the trace link is. The number of different activated lines of code indicates the weight of trace links, which represent how much a test case is covered by a source code module and, inversely, how much a source code module is responsible for processing that test case.

Considering that we have trace links from the FP transactional model to test cases and from test cases to source code modules, we apply the property of transitivity achieving indirect trace links between the FP transactional model and the source code. The weight of an indirect trace link is derived by summing weights of all test cases-to-code links which belong to a specific indirect trace link.

4. CASE STUDIES

We conducted three case studies to ascertain the feasibility of our proposed technique. The first case study was limited to FP data functions traceability while the two following case studies were applied for both data and transactional functions.

The first case study, described in section 4.1, evaluated the open-source ERP and CRM Compiere. The second, presented in section 4.2, evaluated the open-source LMS Moodle. The third, presented in section 4.3, evaluated PPGI/UNIRIO's Online Subscription System. We had access to the source code of each of these systems.

The choice for open-source systems on the first and second case studies was made because of the requirement to access their source code and to allow easy replication of our case studies. Nevertheless, we are convinced that the evaluation of non open-source systems is needed to generalize the conclusions concerning the feasibility of our proposed technique. A step in this way is the third case study.

4.1 Compiere

The initial goal of the case study on Compiere was to observe how its functionality had been implemented in the source code over time, based on an FPA point of view. Later, this case study has led to the development of our proposed technique.

The source code of Compiere was downloaded from its website in November 7, 2008. We have made a checkout from its SVN repository, retrieving 5,015 files of which 2,277 were source code modules in Java. We defined the granularity of source code traces as a module (single file) for this case study.

We did not find an FP data model describing Compiere. Thus, the set of its ILFs was obtained by reverse engineering from its Postgres database creation script. The reverse engineering process involved identifying tables which seem to represent domain concepts and aggregate them in logically related groups of data. The complete database schema contained 465 tables. After discarding tables which did not represent domain concepts (like those representing a database error log and conversions units), the database schema was reduced to a set of 211 relevant tables. By analyzing table names, we have found 162 ILFs. Each ILF was manually traced to a set of tables, making up a set of 211 trace links between the 162 ILFs and the 211 relevant tables.

To relate data functions to source code modules, we selected only the 1,667 source code modules from the core of the system. Support modules (like Jakarta ECS and LDAP authentication) and testing scripts, among others, were discarded. References to tables which were formerly traced to ILFs in the preceding step were sought in the selected source code modules. We found 33,523 references to database tables in 700 Java source files. Suppressing duplicate references between the same table and file, we found 2,608 distinct trace links between 83 tables and the previous 700 files.

We understood that each source code module should have one ILF that could be considered as the main data handled by the module. As each ILF was traced to a set of tables, we counted the number of references to ILFs in each source code module, based on trace links between tables and modules. The most referenced ILF of a module was considered as its main ILF. When we found two or more ILFs as the most referenced ones (both with the same number of references), one of them was chosen as the main module based on name and content of the module.

By transitivity, a set of 700 indirect trace links between 76 ILFs and the 700 source code modules was achieved. The 86 ILFs which were not traced made us rethink the trace links between ILFs and source code, along with the source code modules which could be discarded from the reference search process. The proposed approach was fixed to account for this limitation and reevaluated in the following case study.

4.2 Moodle

The main goals of the case study on Moodle were to evaluate solutions to issues faced in the previous case study and to develop a strategy for generating transactional function trace links. Furthermore, we applied the technique to a different system – a step toward its generalization.

The source code of Moodle was downloaded from SourceForge (http://sourceforge.net) in June 10, 2009. We have made a checkout from its CVS repository, retrieving 30,960 files of which 10,170 were source code files in PHP. We defined the granularity of source code traces as a module for this case study.

We could not find an FP data model describing Moodle. Thus, the set of its ILFs was obtained by reverse engineering from its MySQL database schema. The reverse engineering process involved identifying tables which seemed to represent domain concepts, aggregating them in logically related groups of data, and conducting an ad hoc execution of the system to confirm if they were effectively used by the available functionalities. Testing is also used by [8] to extract the system's conceptual model as a class diagram. The initial set of 198 tables was analyzed, leading to 28 ILFs. Each ILF was manually traced to a set of tables, making up a set of 108 trace links between the 28 ILFs and 108 tables. Other 90 tables could not be traced to meaningful ILFs.

Moodle design isolates data handling in a single source code module that provides a set of generic methods to access database tables. All other source code modules are supposed to use these methods to query and update data. Thus, we sought for references to these methods instead of searching for direct references to data, identifying the name of the accessed table (one of the parameters for functions provided by the data access module).

This time we had not discarded modules, considering that every file which handles a table needs to be traced to it. We found 77,981 references to tables in 1,403 PHP source code modules. Discarding duplicate references between the same table and module, we found 4,024 distinct trace links between 192 tables and the previous 1,403 source code modules.

Next, we evaluated three policies to generate the weight for the links between ILFs and source code modules: (a) the most referenced ILF is traced to the module (the same of Compiere); (b) all referenced tables are traced to the module without weights; and (c) referenced tables are traced to the modules proportionally to the number of references.

After evaluating the generated traces for the three policies, we concluded that the first and the second policies were dominated by the third. The first incurred in strong loss of links, while the second was unable to represent how much the table is linked to the each source code module. We selected the third policy because it represented trace links for every table and allowed us to understand that some links were stronger than the others. For this policy, a set of 2,076 indirect trace links between the 28 ILFs and 1,316 source code modules was achieved. In the end, all ILFs were traced to source code.

The creation of links to transactional functions began with the download of another release of Moodle (version 1.9.7) in February 11, 2010 (we were conducting an exploratory study for eight months). We retrieved 5,169 files, of which 2,346 were source code files in PHP.

The reverse engineering of Moodle transactional functions involved testing the system, looking for transactions offered though commands available in the end-user interface. Due to the volume of distinct transactions provided by Moodle, we restricted our analysis to the *user admin* subsystem. We found 35 transactional functions in this subsystem. We developed 43 test cases using Selenium IDE (http://seleniumhq.org/projects/ide/), a tool which allows the automation of test cases. In addition, each EI, EO, and EQ was manually traced to a set of test cases, making up a set of 42 trace links between the 35 transactional functions

and 42 test cases. A single non-traced test case, the "Logout" operation, is lacking because it cannot be considered a transactional function, since it does not read or change data in domain tables.

To collect code coverage information while executing the test cases, we used the XDebug tool (http://www.xdebug.org). This tool is installed as a module on the web server and identifies every single line interpreted by the execution of code using a functionality called *function trace*. Test cases were executed one by one while XDebug collected logs with activated files, and source code lines, and time. At the end of each test case, log files were moved to a distinct directory. We developed a script to analyze these logs and extract the number of single (non-repeated) lines of code executed during each test case. We found 2,698 trace links between 43 test cases and 252 source code modules. A set of 2,666 indirect trace links between the 35 transactional functions and 251 source code files was achieved.

4.3 PPGI/UNIRIO's Online Subscription System

The main goal of the case study on PPGI/UNIRIO's Online Subscription System (further, we will refer to the system as PPGI-OSS) was to apply the final version of our technique, strictly as described in Section 3. Furthermore, we applied the technique to a different system – a step toward its generalization.

The access to PPGI-OSS's source code was allowed by its developer. We received it in April 15, 2010, retrieving 67 files, of which 42 were source code files in PHP and one MySQL database creation script. We defined the granularity of source code traces as a module for this case study.

The developer did not produce an FP data model describing PPGI-OSS. Thus, the ILF set was obtained by reverse engineering its MySQL database creation script and interviewing the developer. The reverse engineering process involved identifying tables that seemed to represent domain concepts, aggregating them in logically related groups of data, and conducting an ad hoc execution of the system to confirm if they were effectively accessed by provided functionalities. The initial set of 8 tables was grouped into 6 ILFs. Each ILF was manually traced to a set of tables, making up a set of 8 trace links. All tables were traced to ILFs.

References to tables were sought in the source code modules. We found 51 references to database tables in 11 PHP source code files. By suppressing duplicate references between the same table and file, we found 22 distinct trace links between the 8 tables and the previous 11 files. By transitivity, a set of 20 indirect trace links between the 6 ILFs and the 11 source code modules was achieved. In the end, all ILFs were traced to source code.

The creation of links to transactional functions began by reverse engineering of PPGI-OSS transactional functions. This process involved executing transactions offered by the system through commands available in its end-user interface. We found 17 transactional functions and confirmed them by interviewing the developer. Also, 22 test cases were developed and automated to run these transactions using Selenium IDE. Each EI, EO, and EQ was manually traced to a set of test cases, making up a set of 20 trace links between the 17 transactional functions and 20 test cases. The non-traced test cases – related to "Show home page (login form)" and "Logout" operations – are lacking because they cannot be considered transactional functions, since they do not read or change data in domain tables.

To collect code coverage information while executing the test cases, we again used the XDebug tool. Test cases were executed sequentially while XDebug collected logs registering the activated files and source code lines. At the end of each test case, log files were moved to a distinct directory. We developed a script to analyze these logs and extract the number of single (non-repeated) lines of code executed during each test case. We found 212 trace links between 22 test cases and 37 files. A set of 175 indirect trace links between the 17 transactional functions and 36 source code files were achieved.

4.4 Discussion

Key indicators collected from the two last case studies (Moodle and PPGI-OSS) are presented in Table 1. Compiere results are not presented because the first case study used an embryonic version of the proposed technique and was restricted to analyzing the system data functions.

Table 1: Case studies summary

Key Indicators	Moodle	PPGI-OSS
Source codes traced to data functions	1,316 (13%)	11 (26%)
Data functions traced to source code	28 (100%)	6 (100%)
Source codes traced to trans. functions	251 (11%)	36 (86%)
Trans. functions traced to source codes	35 (100%)	17 (100%)

From the former table, we notice that all data and transactional functions were traced to source code modules, but just a few source code modules were traced to data functions (in both case studies) and transactional functions (for Moodle).

Few modules were traced to data functions because both systems provide a data abstraction layer, that is, a limited number of source code modules are responsible for collecting data from the storage system and making these data available for the whole system. Therefore, only these modules could be traced to data functions, though many others could be affected by changes in the structure of the data.

A low percentage of source code modules were traced to transactional functions on Moodle, in contrast to PPGI-OSS, as only a few observed transactions provided by Moodle (the user administration subsystem) were effectively covered by the available test scenarios. Therefore, the proposed technique depends heavily on the availability of test scenarios to deliver a more complete set of trace links.

5. PROTOTYPE

This section presents the FPT Plug-in (Function Point Tracing Plug-in), the prototype of a traceability tool that attempts to demonstrate how trace links created by our technique could be shown in daily-work tools used by the development team.

Potential traceability users include software architects and developers. Both groups do most of their work through an Integrated Development Environment (IDE). Our prototype collects information about a project being developed in such environment and presents its source code modules and their relationships with FP data and transactional functions. It indicates which functions are related to a given source code module and

which modules are related to a given function. Our intention was to improve program comprehension and support maintenance by providing a way to analyze the source code modules which might be affected by a change.

The FPT plug-in was developed as an extension of the Eclipse IDE, a software development environment widely used for Java programming but also compatible with other languages, such as C, C++, and PHP. The plug-in user interface comprises two views: FPT Navigator and FPT File Details.

The FPT Navigator view presents two options – "Data Functions" and "Transactional Functions" – under which the functions composing the FP model and their related source code files are listed. It also presents the "Traced Files" option, which lists files traced for some data or transactional function, thus providing a two-way navigation system.

The "Data Functions" option shows the list of all system data functions. By selecting one of these functions, two options are made available: (a) the option "All Files" lists all files related to selected data function; and (b) the list of the transactional functions related to the selected data function. By selecting one of these transactional functions the navigator presents a list of source code modules related to both the selected data and transactional functions. Modules are presented in descending order, according to the weight of their trace link to the function under interest.

The "Transactional Function" option shows the list of all system transactional functions. By selecting one of these functions, two options are made available: (a) the option "All Files", which lists all files related to selected transactional function; and (b) the list of all data functions related to the selected transactional function. By selecting one of these data functions, the navigator presents a list of source code modules related to both the selected transactional and data function. As with the data functions, modules related to transactional functions are presented in descending order, according to the weight of their trace link.

The FPT File Details view has traceability information about the currently active file in the IDE. All data and transactional functions related to this file are presented in descending order of weight of their trace link.

We believe that the plug-in is useful in situations, such as: (a) a developer trying to understand source code structure and provided functionality uses the FPT Navigator to find out which module implements a given functionality; (b) a developer adding a field in some data function uses the FPT Navigator to identify source code files which handle this data function; and (c) an architect investigating performance issues of an entire system finds a source code module with low performance. FPT File Details allows realizing that this module is used by several transactions. By fixing the module issue, the entire system seems to have recovered normal performance. The conclusion is that module performance issue was the root cause of the system issue.

The prototype received trace links from Moodle and PPGI/UNIRIO's Online Subscription System case studies. The ad hoc usage of FPT Navigator demonstrated that most related files (considering the decreasing order by weight) usually had similar names to their related data and transactional functions. In the same way, FPT File Details demonstrated that most related data and transactional functions had similar names to their related files. It seems to demonstrate that the number of references to tables (for data functions), enacted lines of code (for transactional functions) and the idea of weighting trace links could help to improve software development scenario.

6. THREATS TO VALIDITY

Although case studies have been conducted and a prototype was developed to use information gathered from applying our proposed technique, there are still threats to its validity, as we shall discuss in this section.

The case studies have shown that a low percentage of source code modules were traced to tables. In a multi-tier architecture, table references cannot be found outside the data access layer. Thus, our approach misses trace links for source code modules of other layers, such as business rules and presentation. This scenario poses difficulties to a more precise identification of modules which might be affected by changes in data functions.

Although we focus on IS developed using FPA, the systems used in the proposed case studies were open source and in-house projects, which did not have FPA models. The lack of real FPA models is a larger threat for the first and second case studies. However, in the third case study, we had this issue controlled by producing an FPA model according to observed transactions and thereafter validating the model with the system's sole developer.

Formal and automated test cases are needed to ensure quality on high scale and complex systems. Our technique was produced to enable these systems to achieve traceability benefits only if they have high test coverage. Having a large set of requirement-based test scenarios is a concern for the practical use of our proposal. While common sense dictates that having such test scenarios is healthy for a software development project, we acknowledge that many IS project might be deployed without such artifacts.

Finally, the first stages of the proposed approach rely heavily on human input. The subsequent process combines these data with information collected through static and dynamically analysis. Therefore, final results are sensible to mistakes in the early provided input. This includes issues such as incorrect modeling, outdated requirements, and low code coverage of test scenarios.

7. RELATED WORK

Information Retrieval (IR) is used by [2] to discover traceability links between free-text documentation produced during development and maintenance cycle and the source code. Documentation is analyzed by the following steps: (a) letter transformation, where all letters in a document are converted to lower case; (b) stop-word removal, where articles, punctuation, and other irrelevant characters are removed; and (c) morphological analysis, where plurals are converted to singulars and conjugated verbs to infinitives. Source code is analyzed by the following steps: (a) identifier extraction, where identifiers are extracted from the code; (b) identifier separation, where identifiers composed of two or more words are split into separate words; and (c) text normalization, where the three steps for documentation are applied upon the former identifiers. Hereafter, they are passed through indexers and are classified. The work tested two IR models for classification: probabilistic and vector space models. It forms a theoretical basis to our work as it highlights traceability benefits. In addition, our work parses source code to recover traceability links.

LeanArt tool [7] generates trace links between use case diagrams elements (actors and use cases) and source code elements (methods and variables). After some links are inserted manually, a machine-learning algorithm compares the names of use cases and source code elements. In addition, a second machine-learning algorithm compares and follows the values of variables during runtime. These algorithms discover new trace links and detect false ones. The goal of generating links between source code elements and use cases (somewhat related to transactional functions in IS, as both represent data handling processes) renders this approach related to the one presented in this paper.

TraceAnalyzer tool [6] uses information generated from usage scenarios (i.e. test cases) to generate traceability links between: (a) scenarios and source code; (b) model (e.g. class diagrams) elements and source code; (c) scenarios and model elements; and (d) among model elements. The first activity is *hypothesizing,* where some links must be manually identified from documentation. The second activity is *atomizing*, where scenarios are monitored by a code coverage tool and a set of links is built, namely a *footprint graph*. The third activity is *generalizing*, where the graph is traversed from its leaves to their parents. The fourth activity is *refining*, where the graph is traversed from its root to its leaves. These major activities yield the final set of trace links. This technique, known as Scenario-Based approach, forms the basis for our transactional function traceability, which also uses information from code coverage tools.

8. CONCLUSIONS

This paper presented an approach to generate trace links between function points and source code. These links connect low-level implementation to requirements, supporting program comprehension, maintenance, and impact analysis. The proposed approach comprises two steps, aligned to the division of a function-point model: data function traceability and transactional function traceability. Its feasibility was tested through three case studies and the potential for using the produced trace links was addressed by developing a tool prototype supporting the maintenance of a software project within a well-known programming environment.

Contributions of this work include: (a) definition and development of a technique to generate trace links between a function point model and the source code; (b) definition, planning, execution, and analysis of three case studies using three software systems of distinct sizes and origins and written in two different programming languages; and (c) development of a tool prototype using results of the trace link generation technique.

Some limitations of the proposed approach include: (a) data-access abstraction layers (such as web services, stored procedures, views, or object-relational mappings) can cause deviations on data functions traceability by obscuring table references in the code; (b) data from external systems is not traced, since the current definition of the proposed technique does not support EIFs; (c) the quality of trace links directly depends on the quality of manually provided input, such as the mapping between relational tables and data groups; and (d) the need for manual input may be prohibitive, specially on large scale systems.

Future research may address some of these limitations by (a) performing case studies in a large set of systems, preferably with real FP models; (b) evaluating *precision* and *recall* metrics in comparison with other traceability techniques (c) assuring that traceability benefits are actually achieved through technique trace links; (d) evaluating the effectiveness of counting references to tables for gathering references to data functions; and (e) reducing the dependence on manually-provided input.

9. ACKNOWLEDGEMENTS

This work is funded by CNPq (*Conselho Nacional de Desenvolvimento Científico e Tecnológico*) and FAPERJ (*Fundação Carlos Chagas Filho de Amparo à Pesquisa do Estado do Rio de Janeiro*). The authors would like to thank the agencies for their support.

10. REFERENCES

[1] Gotel, O. C. Z., and Finkelstein, C. W. 1994. An analysis of the requirements traceability problem. In *Proc. of the IEEE Int'l Conference on Requirements Engineering* (ICRE'94). IEEE Computer Society Press, 94-101.

[2] Antoniol, G., Canfora, A., Casazza, G., De Lucia, A., and Merlo, E. 2002. Recovering Traceability Links between Code and Documentation. *IEEE Trans. Softw. Eng*. 28, 10 (Oct. 2002), 970-983.

[3] Kannenberg, A., and Saiedian, H. 2009. Why Software Requirements Traceability Remains a Challenge. *The Journal of Defense Softw. Eng*. 22, 5 (Jul/Ago. 2009), 14-19.

[4] Albrecht, A. J. 1979. Measuring application development productivity. In *Proc. of Joint Share, Guide and IBM Application Development Symposium* (Monterey, CA, USA, Oct. 1979). IBM Press. 83-92.

[5] Egyed, A., Biffl, S., Heindl, M., and Grünbacher, P. 2005. A value-based approach for understanding cost-benefit trade-offs during automated software traceability. In *Proc. of the 3rd Int'l Workshop on Traceability in Emerging Forms of Software Engineering* (TEFSE '05). ACM, NY, USA, 2-7.

[6] Egyed, A. (2001). A scenario-driven approach to traceability. In *Proc. of the 23rd Int'l Conference on Software Engineering* (ICSE '01). IEEE Computer Society, Washington, DC, USA, 123-132.

[7] Grechanik, M., McKinley, K. S., and Perry, D. E. 2007. Recovering and using use-case-diagram-to-source-code traceability links. In *Proc. of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (ESEC-FSE '07). ACM, NY, USA, 95-104.

[8] Tan, H. B. K., Zhao, Y., and Zhang, H. 2009. Conceptual Data Model-Based Software Size Estimation for Information Systems. *ACM Trans. Softw. Eng. Methodol.* 19, 2, Article 4 (Oct. 2009), 37 pages.