Using Dependency Information to Select the Test Focus in the Integration Testing Process

Lars Borner
Institute for Computer Science
University of Heidelberg
Im Neuenheimer Feld 326
Heidelberg, Germany
borner@informatik.uni-heidelberg.de

Barbara Paech
Institute for Computer Science
University of Heidelberg
Im Neuenheimer Feld 326
Heidelberg, Germany
paech@informatik.uni-heidelberg.de

Abstract—**Existing software systems consist of thousands of software components realizing countless requirements. To fulfill these requirements, components have to interact with or depend on each other. The goal of the integration testing process is to test that the interactions between these components are correctly realized. However, it is impossible to test all dependencies because of time and budget constraints. Therefore, error-prone dependencies have to be selected as tests. This paper presents an approach to select the test focus in the integration test process. It uses dependency and error information of previous versions of the system under test. Error-prone dependency properties are identified by statistical approaches and used to select dependencies in the current version of the system. The results of two case studies with real software systems are presented.**

Keywords; integration test, dependency information, test focus selection

I. Introduction

The huge number of software components of today's software systems challenges testers in the unit, integration and system testing process. While the unit testing process addresses the single components and the system testing process checks that the system requirements are correctly realized, the integration testing process focuses on the dependencies between software components. The goal of the integration testing process is to show that the dependencies are realized correctly. "*Integration testing is a search for component faults that cause intercomponent failure.*" ([3], p 629)

In every testing process several decisions have to be made to successfully control, execute and finish the process [5]. One of the main decisions is the test focus selection. The test focus represents the parts of the system that have to be tested more extensively than parts not selected. This is necessary because of the limitations of available resources (time, budget).

To spend these resources wisely, testers have to select the parts of the system that are error-prone, i.e. that have a higher probability to contain an error. Several approaches can be found to predict error-prone components, using information of previous versions of the systems ([2], [10], [11], [14], or [17]). However, these approaches focus on components only. Therefore, they can only be used in the unit testing process, where single software components are tested.

This paper presents a new approach to predict error-prone dependencies in a software system. A dependency is a unidirectional relationship between two components. One component (*dependent component*) depends on the functionality of a second component (*independent component*). A dependency is characterized by several properties. Between two components *A* and *B* only two dependencies can exist, one unidirectional dependency from *A* to *B* and vice versa from *B* to *A*.

In our approach we consider source code files as software components and use dependencies between these files to select the test focus. All dependency properties considered can automatically be extracted from the source code by using a source code analyzer. Previous versions of the system under test are used to identify properties that correlate with the number of defects in the source code files. To uncover correlations between the properties and the number of errors, the statistical test of Kruskal and Wallis [9] is used. Correlations that are persistent in all former versions are used to select the dependencies to be tested in the current version of the system.

The following sections are organized as follows. Section 2 explains the properties that are used to characterize a dependency. Section 3 gives an overview of the whole approach. It is followed by a detailed description of two case studies of real large-sized software systems. In section 5 an overview of existing approaches is given. The last section summarizes the paper and the experiences we gathered in the case studies.

II. Dependency Properties

Our approach uses 13 different properties to characterize a dependency. These properties were gathered from several existing approaches: [7], [8], [13], or [26]. We choose properties that can automatically be extracted with a source code analyzer. In our work we used the open source tool SISSy (Structural Investigation of Software Systems [22]).

This tool analyzes source code files of several programming languages and exports an abstract model of the source code into a data base. The required information about dependencies and their properties can be extracted from the model by simply using SQL-Queries. For this, we use a small self-developed tool that manages the queries of the database and creates a file that can be used in the statistical tool SPSS [23].

In our research work we focus on source code files because several approaches exist to determine the number of errors per file [15], [19] which is required to identify the most error-prone dependency properties. The properties of dependencies between source code files are based on dependencies between classes in an object-oriented system. The dependencies between classes have to be mapped to dependencies between files. This is necessary because a source code file can contain more than one class. The mapping adheres to the following mapping rules:

Rule - Inheritance: A dependent file *A* inherits from an independent file *B,* if at least one class in *A* is a sub-class of a class in file *B*.

Rule - Service Call: A dependent file *A* calls at least one service of an independent file *B,* if at least one class in *A* calls at least one method of a class in *B*.

Rule - Attribute Access: A dependent file *A* accesses at least one attribute of an independent file *B,* if at least one class of file *A* accesses at least one attribute in the file *B*.

Using these mapping rules, several properties of dependencies between source code files can be used in our approach. The properties can be divided into two main categories: inheritance and client/server.

A. Inheritance

A dependency with inheritance properties exists, if there is at least one class in the dependent file that is a sub-class of a class in the independent file. Three modifications in an inheritance dependency are interesting for the integration test process: adding new services, adding new attributes, and overriding existing services. In all three cases the sub-class changes the super class by adding a new functionality and/or by modifying existing services. This is important, because the concept of sub-typing ([13]) allows the use of a sub-class in the source code where a super class is expected. If the sub-class changes the behavior of the super class, it can lead to an error. Therefore, it is important to look for correlations between the number of errors and the amount of modification. The following properties are used. The *number of new attributes* indicates how many attributes are added in sub-classes in the dependent file.

The number of new services indicates how many services are added in the sub-classes in the dependent file and the *number of overridden service* indicates how many services of the super classes in the independent file are overridden by the sub-classes in the dependent file.

B. Client/Server

A client/server dependency exists, if at least one class in the dependent file accesses attributes and/or calls services of at least one class in the independent file. Several properties can be interesting for the integration tester, because they can give hints on possible sources of error. The *number of accessed attributes* represents the number of attributes that are accessed by the classes in the dependent file. Every attribute is counted only once, even if it is accessed more than once. Accessing an attribute of a foreign class can influence the behavior of the class and may lead to an inconsistent behavior in one or both classes. The *number of services called* indicates how many services of classes in the independent file are called by classes in the dependent file. Every service is counted only once even if the service is called more than once. A service call may fail because of different reasons. Possible faults can be found in [4] (p. 162), e.g. “Message sent to wrong supplier”, “Message not in supplier”, or “documentation/code mismatch”. Furthermore, the input and output parameters can be sources of failure. The *number of input parameters* sums up all input parameters that are used in the services called by classes in the dependent file in the classes in the independent file. As stated in [16] an input parameter may be misinterpreted by the server (the called class). The *number of services with at least one input parameter* represents the number of all services called by the client that uses at least one input parameter. The *number of complex input parameter* sums up the number of complex input parameters that are used in the services called by the classes in the dependent file. An input parameter is complex if it is a class itself and not a simple data type like Integer or Character (see Java programming language). A complex parameter can encapsulate several states, and changing a state may influence the behavior of the calling client, the server or the parameter object itself. The *number of services with complex output* indicates the number of services called by the client that return no simple results. This complex result may be misinterpreted by the client, which will lead to a failure. The *number of services with parameters of the same type* comprises all services called by the client that have at least two parameters of the same type. Calling a method with at least two input parameters of the same kind can lead to an error, as the parameter may be swapped at developing time without a compiling error. However, at runtime the swapped parameter may lead to a faulty system behavior.

From the properties already presented, three new properties can be computed that cannot be extracted directly from the source code: *average number of input parameters per service call*, *average number of complex input parameters per service call*, and *percentage of service calls with complex output*. The first one is computed by dividing the number of input parameters by the number of services called. The second one is computed by dividing the number of complex input parameters by the number of services called and the last one is computed by the number of services with complex output divided by the number of services called.

III. TEST FOCUS SELECTION

Our approach to test focus selection for the integration testing process is based on information from previous versions of the system. An overview of all steps that have to be performed can be found in Figure 1. All steps are necessary, because "*There is no universal metric or prediction model that applies to all projects*" [12], i.e. there are no dependency properties that can be used in all software systems to select the test focus. Correlations that were identified for a software system A and its previous versions do not have to exist in a software system B.

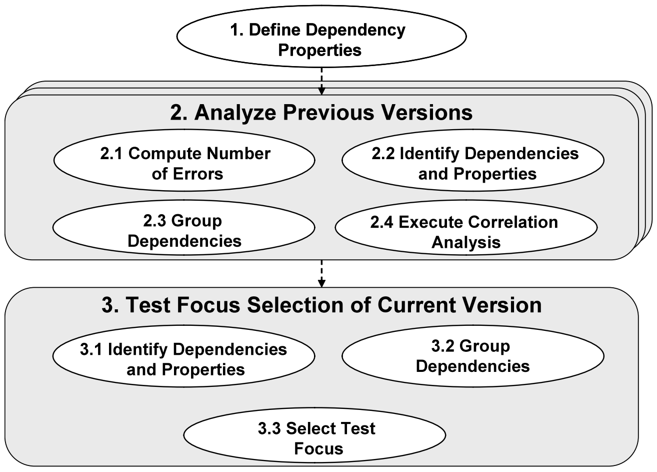
1. Define Dependency Properties
2. Analyze Previous Versions
2.1 Compute Number of Errors
2.2 Identify Dependencies and Properties
2.3 Group Dependencies
2.4 Execute Correlation Analysis
3. Test Focus Selection of Current Version
3.1 Identify Dependencies and Properties
3.2 Group Dependencies
3.3 Select Test Focus

Figure 1 Test focus selection approach

Our approach is divided into three main steps. First of all, the dependency properties that are to be used in the correlation analysis have to be identified. In our case studies we use 13 different properties to characterize a dependency. All 13 properties can be easily extracted from the source code of the system under test. However, it is also possibly to use more or less than these 13 properties. Second, the previous versions of the system have to be analyzed and finally, the identified correlations are used to select the test focus of the current version.

A. *Analyzing Previous Versions*

In the second main step the previous versions have to be analyzed. For every version to be analyzed, four different sub-steps are performed (see Figure 1). First, the number of errors per file has to be computed. These numbers are later used to identify correlations between these numbers and the dependency properties. In [19] a heuristic approach is introduced to use information of a version control system (e.g. Subversion [24]) and a bug tracking system (e.g. Bugzilla [20]) to determine the number of errors per source code file. Within the bug tracking system uncovered defects are documented. Every defect gets a unique identifier (ID). Every time a developer fixes a defect, he/she has to update at least one source code file. These files are checked into the version control system and a new revision (or version) of the files and the system are automatically created. For every check-in the developer adds a comment to describe what he/she has done, for example "#1234 fixed", where "#1234" is the ID of the defect he/she fixed. Zimmermann et al. propose to search for all defect IDs within the check-in comments to identify all files that had to be changed during the fixes. This information is used to determine the number of errors per file.

In parallel the dependencies of the version and their properties have to be identified. The result of this step is a dependency table containing all dependencies. An example of a dependency table extracted from the source code of the open source tool Eclipse [20] can be seen in Table 1. Every row represents a dependency between two files. The first two columns contain the file name [1] of the dependent, respectively the independent file. The following columns represent the properties of a dependency. The third column for example indicates the number of attributes of the independent file that are accessed by the dependent file and the fourth column contains the number of services of the independent file that are called by the dependent file. This dependency table is extended by the number of errors per file for the dependent and independent file (see last two columns in Table 1). The information about the properties and errors is used in step 2.4 to identify the correlations between them.

TABLE 1 EXAMPLE OF A DEPENDENCY TABLE

Dependent File	Independent File	# Attribute accesses	# Service Calls	Inheritance	…	# Errors in dependent file	# Errors in independent file
GC.java	OS.java	37	75	0	…	2	8
Decorations.java	OS.java	79	72	0	…	5	8
Control.java	OS.java	164	71	0	…	8	8
ASTConverter.java	AST.java	0	63	0	…	3	24
Display.java	OS.java	80	52	0	…	45	8
CodeStream.java	ConstantPool.java	4	51	0	…	2	0
Shell.java	OS.java	49	47	0	…	3	8
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

In our approach the statistical test of Kruskal and Wallis is used to identify correlations. This test is the nonparametric alternative to a one-way ANOVA [9] and is used if the assumption of the one-way ANOVA is not fulfilled. The test of Kruskal and Wallis requires that the independent variable (a property of the dependencies) is ordinal scaled. However, as one can see in Table 1 the dependency properties are ratio scaled. Therefore, we have to transform the ratio scale into an ordinal scale. This is done in step 2.3 by using quantiles [9]. The usage of quantiles enables us to create disjunctive groups of nearly the same size. For every property each dependency is put into one group (quantile). This group contains all dependencies with similar values of this property. For example "Group 1" contains all dependencies, where only one service is called, "Group 2" contains all dependencies where two and three services are called and so

[1] For the sake of readability the full path names of the files are left out.

on. The division of the dependencies into quantiles can be done automatically by a statistic program like SPSS [23].

The results of steps 2.1, 2.2 and 2.3 are used in step 2.4 to identify correlations between the number of errors and the dependency properties. For every property we use the Kruskal-Wallis-H test. We check whether the property has a correlation with the number of errors of the *dependent* and/or the *independent* file. For every group the Kruskal-Wallis-H test computes the average rank based on the number of errors. The higher the averages rank the higher is the number of errors. The average rank for each group can be graphically represented. Such a representation can be seen in Figure 2. It is taken from our case study and represents results of the correlation analysis of the software system Eclipse in version 2.0.

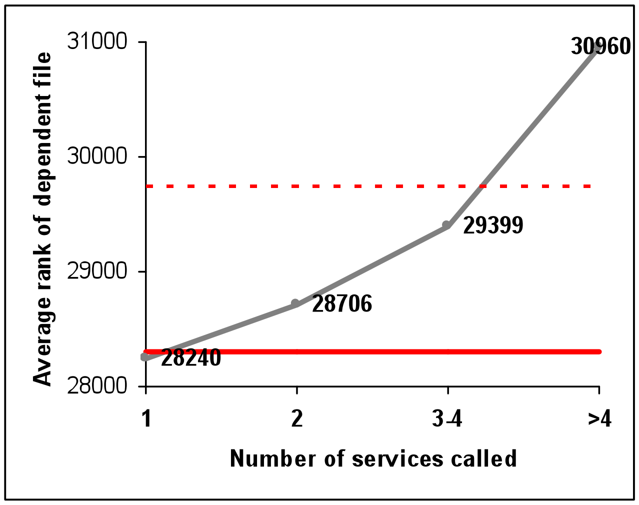

Figure 2 Results of a Kruskal-Wallis-H test (Eclipse 2.0)

This figure shows the average rank in the groups of the property *Number of called services*. As one can see, the first group “1” where only one service is called has the lowest average rank (28,240) and the last group “>4” has the highest average rank (30,960). In our approach we are only interested in the group with the highest average rank, because this group can be used to select the test focus in the later steps. Furthermore, we are interested in groups that range above the average rank of all dependencies (represented by the solid horizontal line in the diagram). If we find the highest group of a property and if it is above the average rank of all dependencies, we have to test whether this group significantly differs from all the other groups of the considered property. For example in Figure 2 we test whether the group “>4” significantly differs from the groups “1”, “2” and “3”. If this is not the case, the correlation found is not statistically significant. A first indicator is the computed significance of the Kruskal-Wallis-H test. It indicates whether at least two groups are significantly different or not. However, if at least two groups differ, the test cannot show which ones. If the test indicates that no groups differ significantly, no statistically significant correlation between the property and the number of errors is found. However, if at least two groups differ, more tests are required. We use the statistical test of Mann and Whitney. This test is applied to two groups and checks whether the groups differ significantly. We apply the Man-Whitney-U test to all combinations of the group having the highest average rank that is above the average rank of all dependencies with all the groups not selected of the property. For the example in Figure 2 we have to perform 3 Mann-Whitney-U tests: for the combination (1; <4), (2;<4) and (3-4;<4). Only if all three tests indicate that the groups significantly differ, we found a correlation between the property and the number of errors[2].

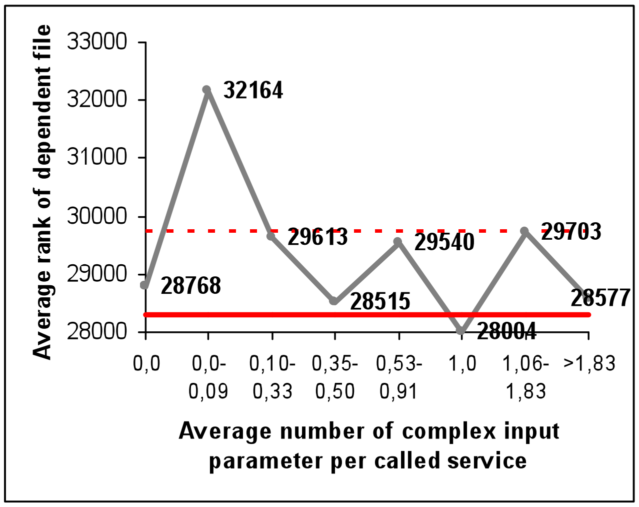

Figure 3 Example of a non-trivial correlation

In our approach we distinguish between two types of correlations: trivial and non-trivial correlations. A trivial correlation is a correlation, where the highest average rank can be found in the highest-valued group (referring to the order of the groups) in respect of the property considered. An example of a trivial correlation is shown in Figure 2. A non-trivial correlation is a correlation, where the group with the highest average rank does not contain the dependencies with the highest values in respect of the property considered. In other words the group with the highest average rank is not the group on the right side of the diagram. An example of a non-trivial correlation can be found in Figure 3. As shown, the second group (0.0-0.09 complex input parameters per service calls) has the highest average rank.

For every property we have to perform the Kruskal-Wallis-H test twice: first, to identify a correlation between the property and the number of errors in the *dependent file* and second, to identify a correlation between the property and the number of errors in the *independent file*.

[2] To reduce the risk of Type I errors, Bonferroni's correction [1] has to be applied.

B. Selecting the Test Focus

The last main step of our approach uses the correlation information gathered in the second main step. A correlation can be used for the test focus selection, if it exists in all of the previous versions. All properties that have a correlation between the numbers of errors of the dependent and/or the independent file (as defined above) are used to select the test focus for the current version of the system. However, to get better prognostic results, the threshold (average rank of all dependencies) has to be increased, because we are interested in the properties that are highly above the average rank of all dependencies. An existing correlation is used for the test focus selection, if and only if the group (according to a property) with the highest average rank is above a given threshold. As we defined earlier, a correlation only exists, if the group with the highest average rank is higher than the average rank of all dependencies. To compute the threshold for the test focus selection, we increase the average rank of all dependencies by a given percentage. In Figure 2 and Figure 3 the dotted horizontal line indicates the new threshold that is 5% higher than the average rank of all dependencies. The new threshold can be computed by the following formula:

```
Average Rank + Average Rank * Percentage
```

If the average rank of the group with the highest rank is above the computed threshold, the corresponding group of the property is used in the test focus selection. As one can see, the property in Figure 2 will be used. The percentage depends on the information about the errors that are available. For example, if there is only a small number of files that contain a small number of errors, the average rank of all groups is very near to the average rank of all dependencies (see case study in section 4.2.). In this case the percentage value has to be very small to compute the required threshold. The following formula can be used to estimate the required percentage value.

$$P_{dep} = 2 * \frac{\max(AR_{prop}) - AR_{dep}}{N_{dep} * N_{prop}}$$

P_{dep} … Percentage value of dependent file
AR_{prop_i}… Average Rank of highest property group
AR_{dep} …Average Rank of all dependencies
Ndep … Number of all dependencies
Nprop … Number of correlating properties

The main idea of the formula is to use the maximal difference of the group with the highest average rank of all correlating properties and the average rank of all dependencies. In our case studies this formula works fine to determine a first estimation for the threshold.

Before we can select a test focus, the dependencies and properties of the current version have to be identified (step 3.1). The result is a dependency table similar to the dependency table represented in Table 1, but without the number of errors per file. In the next step (3.2) the dependencies have to be grouped using the same method as in step 2.3. If we used deciles (dependencies divided into 10 groups) in the previous steps, we also have to use deciles here.

In the last step we have to select all error-prone dependencies as a test focus for the integration testing process.

Definition: *A dependency is error-prone, if it has at least one error-prone property and is assigned to a group with the highest average rank in previous versions.*

Definition: *An error-prone property is a property that correlates with the number of errors of the dependent or independent file in previous versions.*

Using this information, we can assign a test priority to every dependency. A dependency with no error-prone property gets *No Test Focus*. A dependency that has one or more properties that correlate with the number of errors of the dependent files gets *Test Focus Dependent File*. A dependency that has one or more properties that correlate with the number of errors of the independent files gets *Test Focus Independent File*. The test priority *Test Focus Both Files* indicates that the dependency possesses properties that correlate with the number of errors in the dependent <u>and</u> independent file.

The test priority indicates which dependencies should be tested. Moreover, it gives hints on the error location, i.e. whether more errors are likely to be found in the independent, the dependent or in both files.

IV. Case Studies

We applied our approach in two case studies. We chose two real large-sized software projects written in the programming language Java. For both projects we used two previous versions of the system to identify the correlation between the number of errors and the dependency properties. These correlations are used in a third version to select the test focus of this version. In a last step we demonstrate that the selected dependencies are more error-prone than the ones not selected.

The first system we applied our approach to is the open source development tool Eclipse [21]. For this tool the number of defects is available for three versions (2.0, 2.1 and 3.0) in [18]. For this reason we only use two previous versions to identify correlations between errors and properties. The second system the approach was applied to is a commercial tool to manage and monitor financial subventions. This tool does not define versions in the same sense as Eclipse. The tool is continuously enhanced by new functionalities. To select the versions to be used in our case study, we chose three different points in time where large numbers of errors could be found. To compare the results of both case studies, we also chose two versions of the second tool to identify the correlations and to select the test focus for the third version.

A. Eclipse

Eclipse is a Java IDE (Integrated Development Environment) to support developers to create, compile, debug and execute source code. It is realized itself in the programming language Java and can be downloaded from [21]. In our work we analyzed the versions 2.0 and 2.1 to identify the correlations. Version 3.0 was used to select the test focus and to check whether the more error-prone dependencies are selected. A list of defects per file is provided by Zimmermann et al. in [18]. Zimmermann et al. in their work distinguish between two kinds of errors: pre-release and post-release errors. In our work it is not necessary to distinguish between these two types. Therefore, the number of errors is the sum of the pre- and post-release errors.

The Version 2.0 consists of 6,747 source code files and 1,361,739 lines of code (LOC). We identified 56,765 dependencies between these files. Version 2.1 consists of 7,908 files and 1,678,952 LOC. 71,182 dependencies can be identified between these files. In both versions 5.2% of the dependencies have inheritance properties only, 5.7% dependencies have inheritance and Client/Server properties and 89.1% dependencies have Client/Server dependencies only. 2,891 (43%) files of version 2.0 and 2,426 file (31%) of version 2.1 contain at least one error (see [18]).

In a first step we identified the dependencies in version 2.0 and version 2.1. Furthermore, the properties of every dependency were identified and documented by SISSy. We group all dependencies into ten groups (deciles) according to the properties and perform the Kruskal-Wallis-H tests to every property.

As a result we found that in version 2.0 11 properties correlate with the number of errors in the dependent file as defined is section 2.1. Furthermore, we found eight properties that correlate with the number of errors of the independent file. In version 2.1 we identified ten properties that correlate with the number of errors in the dependent file and seven properties that correlate with the number of errors in the independent file. All correlations found are shown in Table 2. The table contains all 13 properties (column 1) and the identified correlations for Eclipse 2.0 (column 2 and 3) and Eclipse 2.1. A positive value *X* in a cell indicates that for a given property the *X*th group has the highest average rank and is above the computed threshold of 5% for version 2.0 respectively 6.2 for version 2.1 and significantly differs from all other groups. A negative value indicates that there is no correlation between this property and the corresponding number of errors. One example: For the property *# Services called* (row 5) and the version 2.0 the group with the highest average rank for the dependent file (column 2) is the group "10". The average rank of this group is above the threshold and significantly differs from all other groups. The property *Average number of input parameters* (row 12) shows no correlation with the number of errors in the dependent file in version 2.0. This is indicated by the value "-1" in the corresponding cell.

A cell with a gray background indicates that a correlation between the property and the number of errors in the dependent/independent file exists in both versions considered, i.e. in both versions the group with the highest average rank is the same. For example in both versions the group with the highest average rank for the property *# Services called* (row 5) is "10" for the dependent file.

In most cases the groups with the highest average rank are the groups that contain all dependencies with the highest values of the given property (mostly group "10"). That means most of the uncovered correlations are trivial correlations. However, there are also non-trivial correlations. A non-trivial correlation is indicated by a positive cell value smaller than "10", for example in row 11. The group with the highest average rank for the property *Average number of complex input parameters* is "4". This group significantly differs from the other groups, has an average rank that is above the threshold and exists in both versions for the dependent and independent file.

TABLE 2 PROPERTY CORRELATIONS (ECLIPSE)

Property	**Eclipse 2.0**		**Eclipse 2.1**	
	Dependent File	Independent File	Dependent File	Independent File
# Overridden services	**10**	**-1**	**10**	**-1**
# New Services	**10**	**-1**	**10**	**-1**
# New Attributes	**10**	**-1**	**-1**	**-1**
# Attribute Accesses	**10**	**10**	**10**	**-1**
# Services called	**10**	**10**	**10**	**10**
# Complex Input Parameter	**10**	**9**	**10**	**-1**
# Input Parameter	**10**	**-1**	**-1**	**10**
# Services with Complex Output	**10**	**10**	**10**	**10**
# Services called with at least one Input parameter	**10**	**10**	**10**	**10**
# Services called with two Parameters of the same type	**10**	**-1**	**-1**	**5**
Average number of complex input parameters	**4**	**4**	**4**	**4**
Average number of input parameters	**-1**	**8**	**-1**	**-1**
Percentage of Services with complex output	**-1**	**5**	**7**	**5**

The information contained in Table 2 is used to select the test focus for Eclipse 3.0. All properties with cells that are highlighted with a gray background are used to select the test focus of version 3.0. In a first step the 96,476 dependencies and their properties are identified. Next, the dependencies are grouped according to their properties. Afterwards those groups of dependencies are selected that had the highest average rank in the previous versions. As a result 1,352 dependencies are selected because they possess properties that correlate with the number of errors in the *dependent file*. 7,737 dependencies possess properties that correlate with the number of errors in the *independent file* and 5,230 dependencies are selected because they possess properties that correlate with the number of errors in *both files*. That means about 16% of all dependencies in version 3.0 are selected as test focus.

To check that we have selected the right dependencies as test focus, we compute the average rank of the selected test focus. Figure 4 represents the average rank of the selected test focus divided by the test priorities *Not Test Focus*, *Test Focus Dependent file*, *Test Focus Independent File* and *Test Focus Both Files*. The average rank is computed for the number of errors in the dependent file (light gray bar) and the number of errors in the independent file (dark gray bar). As one can see, dependencies not selected as test focus have the smallest average ranks. The dependencies selected as *Test Focus Dependent File* have the highest average rank (54,975) of number of errors in the dependent file. The dependencies selected as *Test Focus Independent File* have the highest average rank (56,264) of number of errors in the independent file. Dependencies selected as *Test Focus Both Files* possess an average rank of errors in the dependent and independent file that is above the average rank of dependencies not selected as test focus.

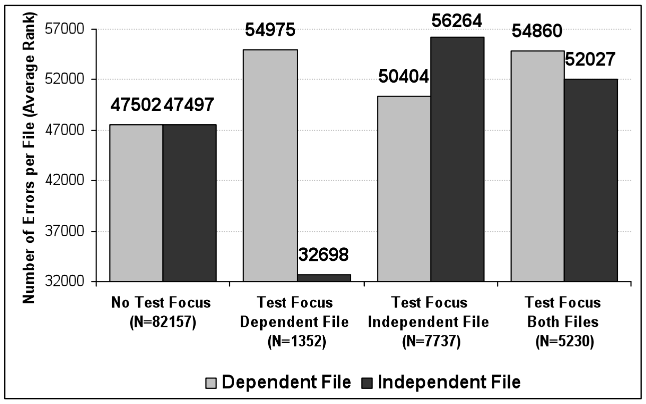

Figure 4 Average rank of selected test focus (Eclipse 3.0)

B. *Subvention Management Tool*

The subvention management tool is a highly distributed software system. It is used to register applications for subventions, offers tool support to check, refuse or accept applications, and monitors the money flow from the government to the companies. It is realized in Java and offers two different graphical user interfaces (Swing, HTML).

It consists of about 23.000 source code files containing more than four million lines of code. It contains about 153,000 dependencies between the source code files: 8,000 dependencies with inheritance properties only, 12,000 dependencies with inheritance and Client/Server properties and 133,000 dependencies with Client/Server properties only.

At the beginning of the case study the number of errors per file had to be determined by using the information of a bug tracking system and the version control system Subversion. The realizing company uses Lotus Notes [25] as the documentation tool for errors. Every error is documented in Lotus Notes and gets a unique ID. If an error is fixed, the developer checks the changed source code files into the control version system. The system creates a new revision number. This revision number is added to the error report by the developer. The revision number is used to identify all files that have to be changed to fix an error. Using this information the number of errors can be determined.

The tool has no fix release cycles and therefore no major releases are defined. We selected three snapshots of the system to perform our case study. These snapshots were used as versions. The first snapshot was from the end of June, the second from the end of July and the third from the middle of September of the same year. The first version contains 602 files (2.62%) that have at least one error, the second 643 files (2.77%). At this time it was unclear whether such a small number of error-prone files could be used to find correlations between properties and the number of errors. However, as shown in Table 3, a lot of correlations were found.

TABLE 3 PROPERTY CORRELATIONS (SUBVENTION)

Property	Version 1		Version 2	
	Dependent File	Independent File	Dependent File	Independent File
# Overridden services	**-1**	**4**	**-1**	**-1**
# New Services	**10**	**-1**	**10**	**-1**
# New Attributes	**-1**	**-1**	**-1**	**-1**
# Attribute Accesses	**-1**	**10**	**-1**	**8**
# Services called	**3**	**-1**	**10**	**-1**
# Complex Input Parameters	**7**	**10**	**-1**	**10**
# Input Parameters	**9**	**10**	**-1**	**10**
# Services with Complex Output	**2**	**-1**	**10**	**2**
# Services called with at least one Input parameter	**6**	**10**	**10**	**10**
# Services called with two Parameters of the same type	**5**	**-1**	**-1**	**10**
Average number of complex input parameters	**8**	**7**	**-1**	**7**
Average number of input parameters	**10**	**-1**	**4**	**-1**
Percentage of Services with complex output	**2**	**4**	**-1**	**4**

In the first version we found ten properties that correlate with the number of errors of the dependent file and seven properties that correlate with the number of errors of the independent file. In the second version five correlations between properties and the number of errors in the dependent file and eight correlations between properties and the number of errors of the independent file were found. However, only a few correlations could be used to select the test focus, because the groups of the correlating properties differ from version to version.

As a result 18.7% of all dependencies in the third version were selected as test focus. 838 dependencies were selected as *Test Focus Both Files*. 27,920 dependencies were selected as *Test Focus Independent File* and 1,373 dependencies are selected as *Test Focus Dependent File*. To check that the test focus selection was correct we also computed the average rank. The results are shown in Figure 5.

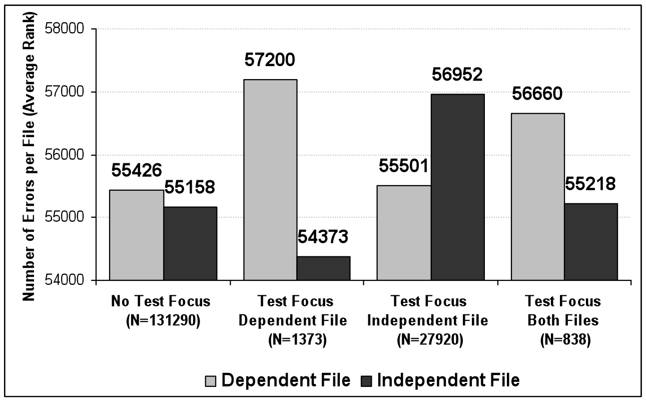

Figure 5 Average rank of selected test focus (Subvention Management Tool)

As one can see, the dependencies with *Test Focus Dependent File* have the highest average rank (57,200) of the number of errors in the dependent file (second light gray bar). Dependencies selected as *Test Focus Independent File* have the highest average rank (56,952) of errors in the independent file. The average rank of dependencies with *Test Focus Both Files* is above the average rank of dependencies not selected as test focus. The small difference between the average rank of dependencies not selected and dependencies with *Test Focus Both Files* according to the number of errors in the independent file can be explained by the small number of all files that contain at least one error.

V. RELATED WORK

In the literature several approaches can be found that deal with defect prediction. They can be divided into three categories. Approaches of the first category try to predict the probability that a file will contain at least one error (e.g. [2], [15]). The second category predicts the exact number of errors that will be contained in a file (e.g. [14], [17]). The last category tries to predict the error density of a file (e.g. [10], [11]). Our approach belongs to the latter category. We identify dependencies with an error density above average

Different statistical tests and approaches can be used to predict errors. Basili et al. uses in [2] the rank correlation coefficient by Spearman to identify correlations. In a second step they use linear regression to predict the probability that a file will contain an error. These statistical tests can only be used to identify trivial correlations. However, we are also interested in non-trivial correlations. Therefore, we use the Kruskal-Wallis-H test to identify the groups with the highest number of errors.

In [18] Zimmerman and Nagappan state that *One drawback of most complexity metrics is that they only focus on single elements, but rarely take the interactions between elements into account* (page 531). They found out that not only the properties (metrics) of one single file should be used to predict errors. In fact the dependencies and their properties are important. However, their approach does not use properties of one single dependency. The authors aggregate the dependency information like the number of clients (dependent files) or the number of servers (number of independent files) within one file and therefore, they only identify correlations between file properties and the number of errors.

None of the existing approaches uses properties of dependencies where a dependency only exists between two files. First ideas about the usage of dependency properties to select the test focus within the integration testing process can be found in [6]. There we use the one-way ANOVA to identify error-prone properties of dependencies. This analysis however cannot be used for every software system because of its assumptions [9]. Therefore the non-parametric Kruskal-Walli-H tests and the Mann-Whitney-U tests should be used as proposed in this paper.

VI. CONCLUSION

In this research paper we introduce an approach to select the test focus in the integration test process. We use information about the test objects of the integration test: the dependencies. We identify correlations between dependency properties and the number of errors of the dependent and independent files in previous versions of the software system under test. The correlations found are used to select the dependencies that have a higher probability to contain errors. The statistical tests used (Kruksal-Wallis-H and Mann-Whitney-U test) can find trivial correlations as well as non-trivial correlations.

Two case studies indicate that our approach works fine. We apply our approach to real large-sized software systems and extract the dependencies between the source code files of the systems. We use information about the number of errors to identify the error-prone properties from two previous versions of the systems. Figure 4 and Figure 5 show that we selected error-prone dependencies. This can be seen by the average rank. Dependencies selected as *Test Focus Dependent File* and *Test Focus Both Files* have a higher average rank of errors in the dependent file (light gray bar) than not selected dependencies. Dependencies selected as *Test Focus Independent File* and *Test Focus Both Files* have a higher average rank of errors in the independent file (dark gray bar) than not selected dependencies.

In our future research work we will focus on the improvement of the test focus selection approach to better predict the error-prone dependencies. We try to combine properties to decrease the number of selected dependencies with a higher number of errors in the dependent and independent files.

Furthermore, the test focus supports the localization of possible errors, because it indicates whether the error is in the dependent file (*test focus dependent file*) or in the independent file (*test focus independent file*).

One important advantage of our approach is the fact that nearly all steps of our approach can be performed automatically. Once the properties to be uncovered are specified, the dependencies and their properties can automatically be extracted by a source code analyzer (like SISSy). The computation of the number of errors per file can also be executed automatically. The statistical tests (Kruskal-Wallis-H, Mann-Whitney-H) are mathematical formula that can easily be implemented.

REFERENCES

[1] Abdi, H.: The Bonferonni and Šidák Corrections for Multiple Comparisons. In: Salkind, N. (Ed.), The encyclopedia of measurement and statistics, Sage, Thousand Oaks, 2007

[2] Basili, V.R., Briand, L. C., Melo, W.L.: A Validation of Object-Oriented Design Metrics as Quality Indicators, IEEE Transaction on Software Engineering, Vol. 22, No. 10, 1996, pp. 751-761

[3] Binder, R., Testing Object-Oriented Systems. Addison-Wesley, 2000

[4] Binder, R., Testing Object-Oriented Software: A Survey, *Journal of Software Testing*, *Verification and Reliability*, Vol. 6, 1996. pp. 125-252,

[5] Borner, L., Illes, T., Paech, B., The Testing Process - A Decision Based Approach, *Proceedings of The Second International Conference on Software Engineering Advances*, Cap Esterel (France), 2007, IEEE Computer Society, pp. 41-49

[6] Borner L., Paech B., Testfokusauswahl im Integrationstestprozess, *Software Engineering,* Kaiserslautern (Germany), 2009, GI, pp. 139-150

[7] Briand, L.C., Feng, J., Labiche, Y., Using Genetic Algorithms and Coupling Measures to Devise Optimal Integration Test Orders, *International Conference on Software Engineering and Knowledge Engineering*, Ischia (Italy), 2002, ACM Press, pp. 43-50

[8] Harrold, M.J., McGregor, J., Incremental Testing of Object-Oriented Class Structures, *International Conference on Software Engineering*, Melbourne (Autralia), 1992, ACM Press, pp. 68-80

[9] Janssen, J., Laatz, W., Statistische Datenanalyse mit SPSS für Windows, Springer Verlag, 2003

[10] Knab, P. Pinyger, M. Bernstein, A., Predicting Defect Densities in Source code Files with Decision Tree Learners, *International Workshop on Mining Software Repositories*, Shanghai (China), 2006, ACM Press, pp. 119-125

[11] Nagappan, N., Ball, T.: Static Analysis Tools as Early Indicators of Pre-Release Defect Density, *International Conference on Software Engineering 2008*, St. Louis (MO, USA), 2008, ACM, pp. 580-586

[12] Nagappan, N., Ball, T., Zeller, A.: Mining metrics to predicts component failures, *International Conference on Software Engineering*, Shanghai (China) 2006, ACM Press, pp. 452-461

[13] Orso, A.: Integration Testing of Object-Oriented Software, PhD Thesis, 1998

[14] Ostrand, T.J., Weyuker, E.J.: How to Measure Success of Fault Prediction Models, *Proceedings of the Fourth international workshop on Software quality assurance 2007*, Dubrovnik (Croatia), September 2007, pp. 25-30

[15] Ratzinger, J. Sigmund, T., Gall,H.C.: On the Relation of Refactoring and Software Defects, *International Working Conference on Mining Software Repositories*, Leipzig (Germany), 2008, ACM Press, pp. 35-38

[16] Suliman, D., Paech, B., Borner, L., Atkinson, C., Brenner, D., Merdes, M., Malaka, R.: The MORABIT Approach to Runtime Component Testing, *Annual International Computer Software and Applications Conference*, Chicago (Illinois, USA), 2006, IEEE Computer Society, pp. 171-176

[17] Zhang, H.: An Initial Study of the Growth of Eclipse Defects, *International Working Conference on Mining Software Repositories*, Leipzig (Germany), 2008, ACM Press, pp. 141-144

[18] Zimmermann, T. and Nagappan, N. 2008. Predicting defects using network analysis on dependency graphs, *International Conference on Software Engineering*, Leipzig (Germany), 2008, ACM Press, pp. 531-540.

[19] Zimmermann, T., Premraj, R., Zeller, A., Predicting Defects for Eclipse, *International Conference on Software Engineering*, Leipzig, 2007, ACM Press, pp. 531-540

[20] Bugzilla, http://www.bugzilla.org/, 2009

[21] Eclipse, www.eclipse.org, 2009

[22] SISSy,http://sissy.fzi.de/SISSy/CMS/index_html, 2009

[23] SPSS, http://www.spss.com/, 2008

[24] Subversion, http://subversion.tigris.org/, 2009

[25] Lotus Notes,http://www-01.ibm.com/software/de/lotus/

[26] http://www.uml.org/