Checking Consistency and Completeness of Software Systems

Harry M. Sneed[1]

Abstract: The following paper presents the current state of the author's research on the subject of static software tracing, a research which dates back to the year 2000 when the author was requested to predict the costs of software maintenance projects. The goal of static tracing is to link the software artifacts produced in software development with the original requirements. These artifacts are linked by comparing the data names they use, i.e. their operands, with the nouns in the requirement text. Artifacts that have data names similar to those used in the requirement texts are considered to be implementations of those requirements.

Keywords: Requirement models, Design models, Code models, Test models, Model comparison, tracing requirements, linking software artifacts, consistency, completeness.

1 Static Checking of System Consistency

One of the goals of static analysis is to uncover consistency errors before a system is delivered. Of course the same errors may also be uncovered by extensive dynamic analysis. But dynamic analysis, i.e. testing, is expensive. It requires the setting up of a test environment and the provision of sufficient test data. In the case of large systems, the execution of several thousands of test cases may be necessary. Even then one cannot be sure that all the inconsistency errors will be found. Besides, there is another reason for checking consistency, namely to determine the degree of product completeness. A software product is only complete when all entities specified are also implemented. This can be ascertained by dynamic analysis if everything is tested but it can also be discovered by static analysis if the artifacts implemented at a lower level are matched against the artifacts specified at a higher level. [Chechik2001]

There are two schools of thought on how consistency should be judged

- the refinement school and
- the verification school

The refinement school requires only backward consistency checks since it assumes that the higher level system description is only a partial description of the lower levels, e.g.

[1] Technische Universität Dresden, Harry.Sneed@T-Online.de

doi:10.18420/se2019-39

the requirement specification only covers part of the design model and the code covers only a part of the design model. The point of step wise refinement is to add details as one progresses from one abstraction level to the next. The problem with this approach is that it will never be possible to verify the code against the specification since the specification is incomplete. Somewhere in the code there is a statement checking the control digit in a bank account number. It was not considered relevant enough to include in the design. Therefore if one is generating test cases from the design –model-based testing – there will be no test of this feature. Unless the tester interprets this as being relevant and adds a test case to cover it, it will never be tested. The test remains incomplete. The refinement approach leaves it to the developer and the tester to decide what is relevant and what is not. The only complete description of the system is the code. In testing one winds up testing the code against itself.

The verification school requires both backward and forward consistency checks since it assumes that the higher level system descriptions are at the same level of abstraction as the lowest level. What distinguishes them is the language. The highest level of description is the natural language followed by the design language and the programming language. There may also be a separate test language. Every last detail contained in the code, details like checking the control digit, appear in the design language. This makes it possible to verify the code against the design and the design against the requirement specification. The test can be fully automated without the need for human interpretation of what is relevant and what not, because the detailed rules are contained in the specification. Backward tracing becomes important. Therefore we must distinguish between entities declared at a lower level and those declared at a higher level. Lower level entities must not necessarily be present in the higher level artefact. This is a sign that the lower level artefact is not complete but it is not inconsistent. Higher level entities that are missing in the lower level artefacts show that the artefacts are not only inconsistent but that the lower level artefacts are incomplete. This is to be interpreted as an error [Lehn2013].

2 Matching Software Artifacts by Names

If development teams were to document the relationships between their artifacts from the very beginning, this matching would not be necessary. However, if that is not done the links between artifacts have to be established after the fact. Matching the names used in the artifacts is one way of achieving this. The basic hypothesis of the approach presented here is that artifacts which contain the same names are related. If a given requirement contains the noun “amount due” it is related to the class which uses the variable “AmountDue”. It can be assumed that this class implements that requirement. The more names that match the stronger that assumption becomes. If only one name matches the association is weak, but if two names match the association becomes stronger. If three names match one can be sure that the two artifacts are related.

Data names appear in all software artifacts. In the requirement texts they appear as nouns in the natural language sentences. In the design model they appear as method and attribute labels. In the code names appear as operands in procedural statements and as labels in data variable declarations. In the database schemas names appear as data field declarations and as keys for searching the data. In the test cases names appear as test data assignments and in test data conditions. Inconsistencies are located by comparing key entities in different artefacts. For instance the classes and variables defined in the design model should be found in the code and the use cases specified in the requirement document should be found in the design model.

3 The Name Matching Algorithm

The main problem in comparing models is that of matching the elements. Elements are identified by name and type. The starting point is the nouns in the requirement text or the change request [AnCD00]. They have to be recognized and collected as depicted in the following use case step description.

03; The <customer> should have the <possibility> of selecting <articles> to be ordered.

04; He must first enter his <customer_number>.

05; Then his <identity> and his <credibility> are checked.

Later the same or similar names are detected in the UML schema <ownedBehaviorxmi:type='uml:StateMachine' name='CustomerCredibility'>

As well as in the code:

```
public CustomerOrder(CustomerOrder cusOrder){
        this.OrderNumber = cusOrder.OrderNumber;
        this.Customer = cusOrder.Customer; }
```

If they can be matched then that particular entity is selected according to its type. Typical types are use cases, classes, methods, interfaces and attributes. Insofar as the types are uniform, i.e. the same types are used in the design as are used in the code and in the test, types can be easily matched. The type names may be a little different but they can be associated with one another. The problem is with the element names. They may differ. For instance, names of classes and attributes in the design model may be written in a different way in the code. Only if the code is generated from the design, can one be sure that they are the same. If the code is written by hand then it is almost for sure that they will not be the same. Thus an algorithm is needed for matching names according to rules.

The first rule of the algorithm is that only names of the same element type are compared. We only compare classes with classes and attributes with attributes.

The second rule is that plural nouns are converted to singular by leaving off the “s” or “es” at the end.

The third rule is that prefixes and suffixes are removed. We should only compare the word stem. Prefixes and suffixes are often used in the code to classify entities of a certain type. Suffixes serve the same purpose. To be able to math these adapted names with the original names the prefixes and suffixes have to be removed. We refer to this as “name stripping”.

The fourth rule is that concatenated names are split up into their surnames. For instance the name “ArticlePrice” is broken down into the two names: “Article” and “Price”. Each sub name is compared separately. If only one of the sub names matches to a name in the other table, the two elements are considered to be related. Experiments have demonstrated that a larger number of false positives may occur here, but it is better to have too many matches than it is to have too few. The human analyst can always delete the false positives.

The fifth rule is that names can be altered to match. If the matching tool recognizes that names almost match but not exactly, it can replace or delete certain characters to force a match. For instance if the word in the requirement document is “Record” and the name in the code “Rec”, the matching tool may only compare the first three characters. Here again there may be many false positives which the human analyst can delete.

Where ever at least one match occurs that entity is considered to be associated with the entity in the other table. At the end of the comparison the user will get a list of all entities for which no match occurred. These entities are then considered to be missing in the other model or document. Of course the human analyst will have to go thru the models to confirm that there entities are really missing, but the job of locating inconsistencies in the models is made much easier and quicker by using the automated comparison. It is also more thorough.

References

[Chechik2001] Chechik, M., Gannon, J.: Automatic Analysis of Consistency between Requirements and Designs. IEEE Transactions on Software Engineering, Vol. 27, Nr. 7, July 2001, p. 651

[Lehn13] Lehnert, S./Farooq, Q./Riebisch, M.: Rule-based Impact Analysis for heterogeneous Software Artifacts”, IEEE Proc.of CSMR2013, Genova, March 2013, p. 209