Code Normal Forms

Markus Pizka
Institut für Informatik
Technische Universität München
Germany - 80290 Munich
pizka@in.tum.de

Abstract

Because of their strong economic impact, complexity and maintainability are among the most widely used terms in software engineering. But, they are also among the most weakly understood! A multitude of software metrics attempts to analyze complexity and a proliferation of different definitions of maintainability can be found in text books and corporate quality guide lines. The trouble is that none of these approaches provides a reliable basis for objectively assessing the ability of a software system to absorb future changes. In contrast to this, relational database theory has successfully solved very similar difficulties through normal forms. In this paper, we transfer the idea of normal forms to code. The approach taken is to introduce semantic dependencies as a foundation for the definition of code normal form criteria.

I. Software Complexity and Maintainability

The complexity of a software system has always been of major interest to software users and developers, because it sheds a light on the costs for future changes. And most software systems, especially successful ones [1], must be changed more or less frequently to keep up with evolving requirements or technical changes. Without these changes a software system would lose its value [2] over time.

Trivially, any software, no matter how weakly or brilliantly it is written can be evolved or changed somehow. The interesting question is the number of resources in time and space needed to accomplish the change. The experience that these costs somehow correlate with properties of the system itself, such as its size, leads to the consideration of the complexity of software systems. Roughly speaking increased complexity stands for greater difficulties in understanding the existing system or parts of it, the implementation of changes taking more time because more components are (or could be) affected, and increased risk for the introduction of new bugs. Therefore, knowing the complexity of a system and means to minimize it are highly desirable.

A. Excessive Variance

In reality, discussing the complexity of software systems has proved itself as being very difficult. One major problem is the enormous variance of costs for different changes within the same system. For example, incorporating a new piece of information into an existing information system maybe rather simple while changing the type of an existing field (e. g. 4 digits instead of 2 for dates) within the same system may require an extensive effort. While various forces influencing software complexity have been broadly studied in research, e. g. [3], [4], [5], the problem of excessive variance persists.

Because of this, the expressiveness of complexity as a property of the system is rather low. We take the stand point that complexity should only be used to map a system s, a change request t, and an organization o to an effort e; $(s, t, o) \rightarrow e$.

B. Maintainability Criteria

Closely related with complexity is maintainability. Here, the term stresses that it is a property of the system independently of the task being performed. Because of their strong economical impact, maintainability issues are regarded in most corporate quality management (QM) systems.

In contrast to complexity, maintainability also encompasses the documentation of the system, the development and maintenance process, and eventually also properties of the organization, such as turnover and knowledge management. Typically, QM handbooks postulate rules that are enforced in reviews, by training developers, and eventually with tool support. Some of these rules may relate to complexity measures while others pertain to non-measurable aspects such as naming of identifiers.

Certainly, the elaborated treatment of maintainability as a quality aspect is useful and goes well beyond regarding the complexity of a system [6]. Complexity considerations are only a part of maintainability. However, there are still strong disagreements on what's maintainable, what's not, and how to achieve it. The reason for this is the absence of a concise and agreed set of criteria.

The dramatic practical effects of this situation are experienced by software developers and user alike. Users are neither able to define the required maintainability nor to assess the quality of the software they paid for. Development organizations, on the other hand, are unable to justify higher costs of better maintainable software (entailing benefits in the long run) or to explain the advantages of their bid over others. Recently, one of our industrial partners summarized these problems by complaining that one of his major customer was now trying to profit from the weak economical situation by only accepting the lowest bids without regarding long-term effects. And there was no solid ground to explain the advantages of an initially more expensive solution.

C. Challenge

The challenge tackled in this paper is the definition of a set of criteria for maintainability that

1) allows objective assessments of the maintainability of software systems and
2) provides a guideline for the systematic construction of software systems with a chosen degree of maintainability.

These criteria need to be formal to be precise, independent of a certain domain or context and free from ambiguity.

Due to the complexity of this question, we for now focus on program code and leave other aspects such as "good" documentation to other research.

D. Approach: Code Normal Forms

Very similar problems as the ones discussed in the context of maintainable code have already been solved long ago in relational database theory (RDBT) by means of normal forms [7]. Though there has been some dispute about the possibilities of anomalies in higher level normal forms [8] the usefulness of normalization has been proven repeatedly [9] and is accepted among practitioners.

Generally speaking, a normalized database scheme guarantees maximum flexibility for future changes to its content as well as to the scheme itself. E. g. if a data element changes, such as the address of a customer in a customer relationship management system, it only needs to be changed in exactly one place. This behavior is exactly what maintainability[1] is aiming at.

The principal idea to achieve it is to systematically eliminate redundancies and inconsistencies. The formal foundation for normalization in RDBT are *functional dependencies*.

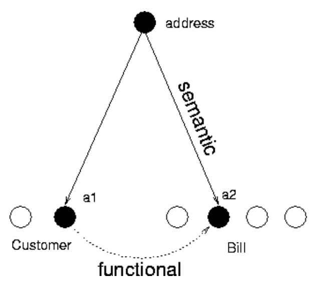

Fig. 1. semantic and functional dependencies

Definition 1: A functional dependency (FD), denoted by $X \rightarrow Y$, between two sets of attributes X and Y that are subsets of the attributes of relation R, specifies that the values in a tuple corresponding to the attributes in Y are uniquely determined by the values corresponding to the attributes in X.

Figure 1 sketches an overly simplified example. The semantic concept *address* is redundantly mapped onto two attributes `a1` and `a2` of the relations `Customer` and `Bill` with a dependency between `a1` and `a2` duplicating the effort for updates of `a1` or `a2`.

In this paper, we want to transfer this idea from data to code by discussing and formulating criteria for normalized code enabling isolated changes. Analogously to FDs, we want to employ semantic dependencies (SD) for the detection and removal of change anomalies.

Outline: Similar to normal forms of data we also need a model of possible changes to code which we will introduce in section III after the discussion of change anomalies of code in section II . These anomalies have their origin in semantic dependencies which are defined in section IV. Based on semantic dependencies, we define normal form rules for code in section V. After comparing our findings with related work in VI this paper concludes with a summary and an outlook on future work.

II. Code Anomalies

Normal forms in relational data base theory aim at eliminating unwanted effects of changes to data causing duplicated effort and loss of integrity. These effects are

[1]except for documentation aiming at program comprehension

called update, insert, and delete anomaly. Expecting familiarity of the reader with these RDBT principles, we only give one short example of a relation causing an update anomaly in table I. An update of a department

employee	dep. no.	dep. name
Chris	5	research
Kirsten	4	training
Peter	5	research

TABLE I. update anomaly

name is awkward, because it must be changed in all tuples representing employees of this department. Storing the department name in a separate relation and only referring to it via the department number would eliminate this anomaly.

Obviously, an update anomaly is unwanted. It causes increased effort for changes (and as a side-effect also increased memory consumption). Now, we want to investigate the question whether there are similar anomalies when changing code.

A. Impact of the Algorithmic Language

When we talk about code change anomalies we are actually referring to the effects of changing algorithms. In contrast to data, algorithms can be specified in numerous different ways. Among the Turing-complete models are the theoretical concepts term and text rewriting systems, turing machine and register-machine programs, mu-recursion, as well as conventional imperative, functional, and logical programming languages.

Although all of these models are Turing-complete they expose different behaviors when it comes to changes of algorithms. Java [10], for example, provides exclusively call-by-value semantics[2]. Because of this, changing a read-only `int` argument into a transient (or read-write) argument may require numerous additional changes of the algorithm.

```
void g(int i) {      void g(int [] i) {    x
 while(i>0 &&...)     while(i[0]>0 &&...) x
  i--;                 i[0]--;             x
}                    }

void f() {             void f() {
  int numiter;         int [] numiter =    x
                        new int [1];       x
  numiter = ...        numiter[0] = ...    x
  g(numiter);          g(numiter);
  ...                  ...
  numiter *= 2;        numiter[0] *= 2;    x
```

Fig. 2. maintainability and call semantics

Figure 2 illustrates an example for this. The left side shows a method `f` calling `g` with an `int` argument specifying the maximum number of loop iterations in `g`. If we want to change `g` so that `i` is a transient parameter returning the actual number of iterations performed, then the type of `numiter` must be changed into an `int[ ]` array. As a consequence, all instructions marked with "x" must also be changed as shown on the right side of the figure. This behavior corresponds to an update anomaly in databases. Clearly one could also change method `g` into a function returning the number of iterations. This, of course, only works as long as there is only a single return value. Increasing the number of return values with the help of a return object will evoke new anomalies, which we will not further elaborate, here. A language providing both call-by-value and call-by-reference semantics, like Pascal does, would allow to eliminate this particular problem.

Hence, the algorithmic language itself has an influence on the maintainability of algorithms. For example, untyped languages with operator overloading such as Smalltalk [11] provide greater flexibility for certain changes than statically typed ones.

To be able to clearly identify the factors influencing the maintainability of algorithms we have to uncouple our view from language specific properties. All following consideration will therefore be based on a model-based algorithmic language. Due to its simplicity and its comparatively high degree of understandability the graphical notation UML *State Diagrams* [12] will be used.

B. Insert Anomaly

An insert anomaly in data base terms means that there is either an unnecessary or unreasonable constraint upon the task of adding new records or adding a new record will inevitably cause unnecessary data redundancy. Very similar phenomena hampering servicing and the evolution of existing software [13] are possible in algorithms, too.

Figure 3 shows an example of an algorithmic insert anomaly. An entry state waits for input and determines the processing sequence consisting of two steps whereby the first step directly activates the second step. The activity of the second step is identical but implemented in two different states X and Y. In this structure the insertion of a new intermediate processing state "new" between the first an the second processing step requires that the state resp. its activity have to be duplicated and redundantly inserted into both sequences.

Replacing the transitions $B \rightarrow X$ and $C \rightarrow Y$ with $B \rightarrow new$ and $C \rightarrow new$ and then branch to X or Y from within new would be an alternative. For this to work, context information had to be passed to new from B and C, and new had to be changed accordingly. Hence, this strategy of insertion also requires more changes than necessary in the ideal case. If the two pairs of processing

[2]Variables for objects in Java are actually references. They are also passed by value.

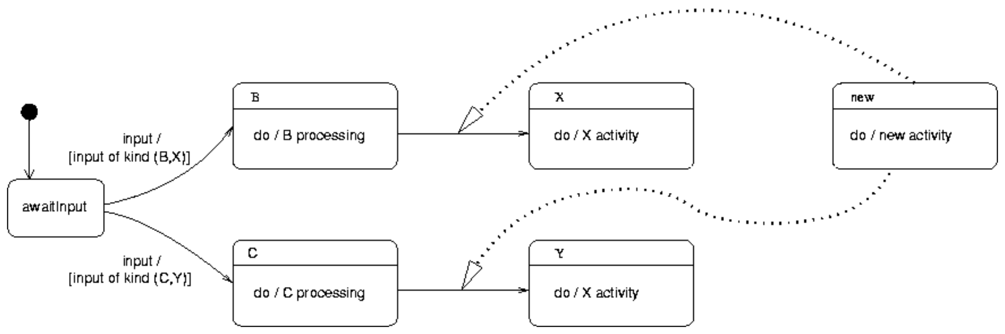

Fig. 3. algorithmic insert anomaly

steps (B,X) and (C,Y) were not directly connected, new could be inserted without these troubles.

In real world programs, statement lists in broad conditionals, e. g. long `switch` statements, often suffer from this anomaly because semantically different concepts are not properly separated but linked through the sequence of statements in each case differentiation. The reason for these unhealthy constructions are usually either performance issues or pressure of time during development. Though performance is without any doubt important, we argue that it should not spoil the structure of algorithms a priori but only be considered in a well directed way, after a maintainable algorithmic structure was developed, first.

C. Update Anomaly

In the context of relational data bases the term update anomaly refers to a situation where modifying a specific value in one record necessitates the same modification in other records and tables. If we replace "value" with a sequence of instructions and "record" respectively "table" with composite states we receive a description of a well-known anomaly of changing algorithms.

Suppose the new activity in the state diagram of figure 3 was inserted into each of the two processing sequences, for example to perform some intermediate processing. As explained above and shown in figure 4 this requires two new states with an identical activity. Now, if we want to change the intermediate processing step according to a new or changed requirement, we have to change it in two places instead of one.

Alongside with a modification requiring identical changes in more than one component, algorithmic update anomalies usually occur in less obvious variations. In most cases it is not the identical change that has to be performed multiple times but variations of the change that need to be applied to different components. The reason for this increased difficulty is that an algorithm can be written in infinite[3] ways without changing its semantics. In contrast to this, representing a certain date in different ways is also possible but a lot less common; e. g. implementing the time of day in the form "hh:mm:ss" or as number of seconds from midnight.

It is also interesting to note, that one anomaly may cause further anomalies. In our example the insert anomaly of the initial structure has now also caused an additional update anomaly. In practice this means that if a weak or unsuitable structure is not rigorously fixed then the system will inevitably lose more and more of its quality until it must be considered unmaintainable and needs to be shut down. This analytical consideration corresponds well with Lehman's laws II and VII of increasing complexity, declining quality, and the need for rigorous additional (re-)work [2].

The update anomaly due to avoidable redundancy has a strong negative impact in practice. Understanding the formation of algorithmic update anomalies and knowing ways how to avoid respectively eliminate them will significantly improve maintenance productivity and quality.

D. Delete Anomaly

A delete anomaly in a data base has the consequence that deleting a record would also remove data not intended for deletion. The mapping of this anomaly to algorithms is a little more subtle than for insert and update anomalies because it depends on the granularity of the delete operation.

In data models delete operations are expected to affect a complete tuple of a relation (or row of a table in technical terms). When we consider deletion of code the unit of the operation is unclear.

If deletion affects a single state in a state diagram, then deletion of a state entails the removal of its complete activity. Now, if the activity consists of a sequence of different operations it is possible that the deletion of the state also

[3]e. g. by adding null operation

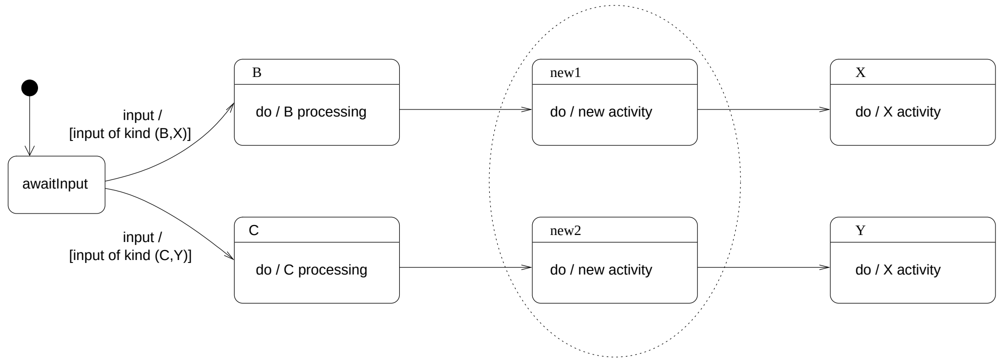

Fig. 4. algorithmic update anomaly

removes operations not intended for deletion. A typical example for this is the combination of an increment operation with a conditional as depicted in figure 5 or contained in a Java statement like `while (count++ <= 42)`. If we

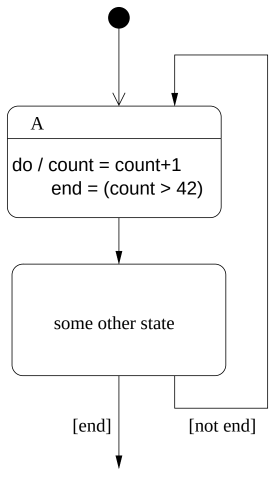

Fig. 5. algorithmic delete anomaly

want to delete the increment of `count` form state A and move it to someplace else, maybe because we first want to perform some calculations with the old value without repeatedly subtracting 1 (which, btw. would cause another update anomaly), we also remove the comparison which was not intended for removal.

The impact and the occurrence of the delete anomaly increase with the granularity of the delete operation. If one considers larger components such as super-states (or methods with extensive statement lists) as units of removal then it becomes evident that careless clustering of different algorithmic concepts into components may cause severe delete anomalies.

III. Maintenance and Evolution Model

Our considerations about code change anomalies show that it is possible to reason about algorithmic change anomalies in a similar way as it is common for relational data structures. In the data base context, the awareness about these anomalies has driven the development of normal forms eliminating the unwanted change behavior. Algorithmic change anomalies could bear a similar chance for the systematic design of maintainable code. But, we have not yet given any reasons why it is reasonable to treat algorithms like data when it comes to modification.

Considerations of change anomalies itself are based on a model of the intended use. For data, the underlying usage model has always been fairly obvious and consists of the dominant operations: read, write (update or insert), and delete. Other operations, such as split and merge of a relation or changing the type of an attribute may also expose anomalous behavior but are not regarded in depth because they are less prevalent. For algorithms the model of use is not as obvious.

A. Usage versus Purpose

In contrast to this, algorithms are usually treated like mathematical functions that are written once and read (executed) multiple times, afterwards. As we can see from the growing field of software maintenance and evolution this perception does not match reality. In fact, algorithms used in software systems do not just represent mathematical functions but reflect some part of the continuously changing real world. Algorithms need to be changed more and more frequently to adapt to changed business processes [14], requirements, or technical innovations. It is well-known that the time and money spent for changing software is about 4 times higher than the effort invested into initial development [15]. This fact justifies a new thought model of the usage of algorithms closely resembling the use of data:

- write operations: insert and multiple updates
- read operation: execution
- delete operation
- nothing is constant; any element may change

This model differs from common maintenance models that classify maintenance tasks according to the purpose of the task. The well-known classification from Lientz and Swanson [16] and variations of it [17] list

- perfective,
- adaptive,
- preventive, and
- corrective

maintenance tasks. This classification is certainly useful for many reasons particular for controlling purposes but it does not point at the intrinsic difficulties of changing code and possible improvements. Our model fills this gap and complements these classifications because any maintenance task, no matter whether it is perfective, adaptive, or other, will require a sequence of the above listed maintenance operations insert, update, and delete. However, focusing on code operations instead of the purpose of the task is a prerequisite to better understand and to improve the maintainability of code.

B. Granularity

Besides the numerous similarities between data and code there are also significant differences that have to be respected. One major difference is the granularity of elements under consideration.

Data base theory is based on relational algebra consisting of sets, relations, and operations on relations. A n-ary relation $R \subseteq R_1 \times \cdots \times R_n$ is simply a subset of the Cartesian product of the n sets R_1 to R_n. R is usually described by a set of n-tuples consisting of n attributes. Now, if we try to reuse the insights gained in data base theory in the context of algorithms then we somehow have to map sets and relations to code. Clearly, elements of sets (or attributes in tables in technical terms) represent the indivisible atoms of the system. Tuples as elements of relations are compounds of these atoms. But, what unit of code forms an atom and what corresponds to the compounds, i. e. relations?

C. Levels of Abstractions

A close look onto data base schemes reveals that relational data models only posses two levels or abstractions — sets and relations. Yes, in data bases there are also views on top of tables but they have no impact on the normalization of a data model. In contrast to this, code stretches large numbers of layers. Figure 6 illustrates this difference by comparing a structured state diagram with data. A super^n state denotes a state that contains at least one super^{n-1} state. A super^0 state is simply an indivisible state. A similar figure could also be drawn for a programming language like Java by stacking instructions, statements, blocks, methods, classes, and packages.

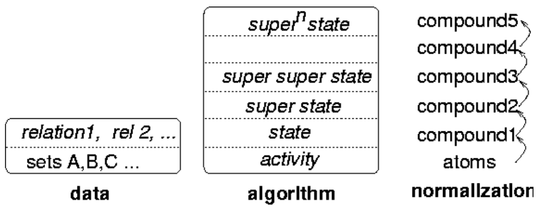

Fig. 6. data and code layers

It can easily be shown that eliminating anomalies on one level of abstraction does not guarantee that higher levels of abstraction are free from anomalies, too. Let's assume a state diagram with two super-states S_1 and S_2 which both implement the same process of a teller machine and a state S that switches arbitrarily between S_1 and S_2. Even if S_1 and S_2 bear no anomalies by themselves the whole system still does because of the redundancy caused by the two semantically equivalent states S_1 and S_2, i. e. none of the two super-states can be changed in isolation.

D. Normalization Strategy

To systematically eliminate all anomalies, all levels of abstraction must be considered. An intuitive strategy to achieve this is to proceed bottom-up. Combining this with normalization in data base theory means that we first have to regard atoms of the code as attributes and normalize them. Then we move up one step and regard compounds of code atoms as relations and normalize them like relations. When moving further up to higher levels of abstraction we continue to treat compounds of growing size as the relations to normalize.

As a matter of fact, normalization in data base theory applies exactly the same strategy. The first normal form (1NF) guarantees that all attributes are atoms. All other normal forms build up on 1NF and deal with the elimination of anomalies among relations which are first level compounds of the atoms. While there are only these two levels in RDBT, algorithms usually stretch numerous levels of abstraction. Without proof, we state that this multilevel normalization can be done efficiently because the degree of overall normalization grows monotonously with higher levels of abstractions. The normalization of a level a remains unaffected by the normalization of level b if $b > a$.

a) Example: An example state diagram is depicted in figure 7 showing a possible expansion of figure 5. Normalization must proceed from inside to outside, that is from the activity in A to states A, B and C, and then to super-state "*some other state*". Note that A and B are themselves compounds that must be expanded and normalized, first.

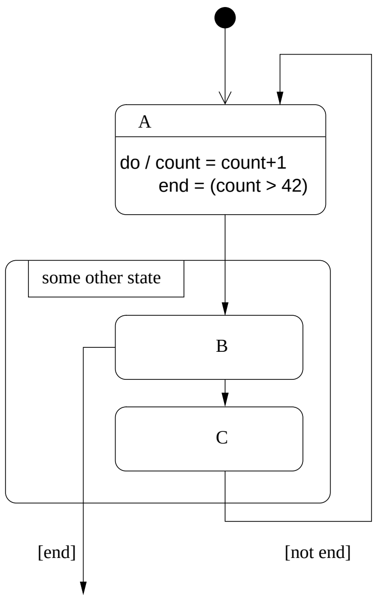

Fig. 7. levels of abstractions

IV. Semantic Dependencies

The substantiation of algorithmic change anomalies and the answer given above concerning the granularity of the code units matching sets and relations in RDBT paves the ground for establishing sound criteria for algorithms without change anomalies.

The definition of these criteria in the data base context relies fundamentally on *functional dependencies* (see definition 1 in section I-D). But how can functional dependencies (FD) be transfered from data to algorithms?

A. Functional Dependencies

In principle, transferring FDs to code appears to be less difficult than expected. A FD $X \to Y$ denotes that the values of the set of attributes Y depend on the values of the set of attributes of X, or short and overly simplified: the values of Y-tuples can be derived uniquely from X-tuples, which is close to a function $\phi : X \to Y$ in the mathematical sense.

Note that FDs are semantic properties of attributes. An FD can only be detected by an expert of the domain who knows the semantics of the attributes. For example, if there are two attributes `City` and `Zip` then we know that they both refer to a similar meaning, combine this with our knowledge that `Zip` is more precise then `City`, and conclude that `Zip` $\to$ `City`. Mathematically, we try to find an abstraction s that maps `Zip` to its semantics and a representation r from the semantics to the representation `City`, so that

$$\forall c \in \texttt{City} : \exists z \in \texttt{Zip} : r(s(\texttt{Zip})) = \texttt{City}.$$

If r exists, then we have found a FD.

If we start to look at algorithms it is useful to first restrict our view to functional programming languages that do not have a global state space. Here we can easily map the idea from FD between data attributes to semantic dependencies (SD) between pairs of functions.

Definition 2: Let $f : A_{in} \to A_{out}$ and $g : B_{in} \to B_{out}$. A semantic dependency, denoted by $f \xrightarrow{s} g$, between f and g specifies that there is a representation function r so that:

$$\forall b \in B_{in} : \exists a \in A_{in} : r(s(f(a))) = g(b).$$

The most trivial SD $f \xrightarrow{s} g$ is the equivalence of two functions f and g. In this case r is s^{-1} and $r \circ s = id$. But, there are also less trivial SDs for example, if g performs a semantically equivalent operation on a different representation, such as f increments integers and g computes increments, too but using hexadecimal strings. Note that the definition of SDs allows to detect trivial as well as rather complex semantic dependencies among functions. We state that SDs are at the core of change anomalies.

The reader might have noticed that FDs consider sets of attributes whereas SDs have for now have only been defined for single functions. At time of writing we have to leave details of the treatment of sets of functions for future work. The idea follows the same approach like for single functions by identifying a subset F of a given set of functions $\mathcal{F}$ whose composed behavior can be reproduced by abstracting and re-representing the composed behavior of a different subset $G \subseteq \mathcal{F}$.

After the introduction of SDs for functions, it is trivial to transfer this concept to algorithms in general which may include a global state space. We only have to regard each algorithm a as a function $a : I \times \Sigma \to \Sigma \times O$, that takes a set of inputs I and a state $\sigma \in \Sigma$ which is an element of the state space Σ and computes a set of outputs O and a new state $\sigma' \in \Sigma$.

However, two tough problems make the use of SDs in practice difficult. First, is the determination of suitable semantic units and second, mutual dependencies between code and data.

B. Semantic Units

It has been mentioned above, that FDs are a semantic property. This is reflected in the definition of SDs by the semantic function s mapping algorithms to their meanings. It is important to note that s and in particular the semantic space used by s can themselves only be a model of the reality. Otherwise, the definition of the atoms in data normalization would have to go down to elementary physics (e. g. breaking zip codes down to digits and their meaning) which would probably lead to a useless universal FD between all elementary particles.

1) Semantic Model Evolution: Therefore, a semantic space is not given but implicitly assumed or explicitly defined by an analyst or software designer. For example, in the data base context one usually considers splitting zip codes and city names into two separate attributes as sufficient to reach first normal formal for this set of attributes. But, this stays only correct as long as we regard zip codes and city names as semantically indivisible units. We usually adhere to this semantic model because it is to assume that zip codes and city names are the smallest units of individual read and write operations. But what happens, if, for some reason, the government decides that every formerly 5 digit zip code starting with a “9” must now be expanded to a 6 digit zip code by inserting a “0” after the first ”9”? This update can obviously not be performed in one place but must be repeated in numerous tuples – which is an update anomaly. This anomaly emerges from the modification of the previously atomic semantic unit `zip code` into two new semantic units `first-digit-of-zip` and `rest-of-zip`. Within this new semantic model, a single attribute `zip code` is no longer in normal form.

Of course, the same problem exists when normalizing algorithms. A modification of the requirements to the system may change the underlying semantic model and cause new anomalies in the previously normalized code which have to be eliminated by a new normalization process based on the new semantic model. But, there are also two good news. First, the re-normalization after a change of the semantic model does usually not require a reorganization of the complete code base but only of those parts whose normal form are violated due to the changed semantics. Second, like in the `zip` and `city` example above, changes of the semantics are only troublesome, if existing concepts are modified or further split. Addition or mergers of semantic concepts have no negative impact.

2) Consequences – Anticipation and Evolvability: Clearly, the true intelligence in the detection of FDs and SDs and the following normalization rests on the definition of suitable semantic units. Furthermore, the ability of the software architect or developer to identify sustainable semantic units is crucial for the evolvability of the system. The boundary between anticipated and unanticipated (expensive ones) changes [18] seems to be defined exactly by the units of the semantic model. Unanticipated changes are those changes that do not fit into the previous semantic model. They can not be accomplished in one place but may cause extensive additional work due to arbitrary new modification anomalies.

The more sustainable the initial semantical model is the more evolvable will the resulting software system be. The identification of concise and sustainable semantic units requires software architects and designer with technical knowledge as well as deep domain expertise and general knowledge.

C. Data and Code Dependencies

The second problem for the identification of proper SDs and FDs in practice besides the semantic model are mutual dependencies between data and algorithms. Any calculation can be implemented in various ways with different fractions coded in the dynamic data state and static code. For example, one can implement a parser [19] of a compiler as a recursive descent parser consisting of extensive static code but hardly any data. But, one could also implement a semantically equivalent parser using a rather simple generic stack automaton code base with a larger dynamic data state.

Because of this, a thorough analysis of SDs must regard algorithms and data in an integrated way by capsulating code and associated data, which can be difficult in realistic programming languages.

V. Normal Forms for Algorithms

After transferring the ingredients of normalization from RDBT to code, such as FDs to SDs and relations to compounds, we can now start to define normal form criteria for code.

A. First Normal Form

The first normal form (1NF) in RDBT requires that each attribute is indivisible. An attribute may neither contain a relation nor multiple values. This can be transfered to code straight forward.

Definition 3: An algorithm is in first normal form iff all of its basic building blocks are indivisible atoms.
The delete anomaly in figure 5 is caused by a violation of this rule. The activity in state `A` can and should be divided into two separate states. But note, that it is up to the architect or the analyst to define the granularity of the basic building blocks. Here, we went down to individual

expressions in activities but we could just as well regard states that do not contain other states as the basic building blocks. It only has to be respected that any modification anomaly at a finer granularity as the selected basic building blocks can not be detected.

B. Higher Level Normal Forms

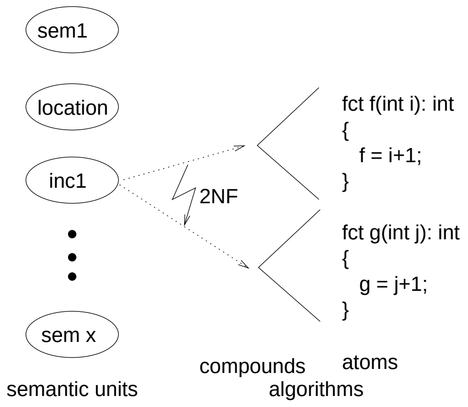

Fig. 8. algorithmic 2NF

The second normal form 2NF for data requires 1NF and irreducible dependency of every non-key attribute on the primary key. This rule can not be transfered directly to code because there are no keys. We run into a similar problem, if we look at 3NF. 3NF requires 2NF and non-transitive dependency of every non-key attribute on the primary key. It seems that the analogy between data and code changes has come to an end, here. But, if we look at the intention behind these criteria at a meta-level then we are able to define not identical but analog new criteria. To some extent, 2NF removes redundancy and 3NF deals with transitive dependencies. In addition to this the multiplicity of layers described in section III-B must be considered. This results in the following 2NF for code.

Definition 4: An algorithmic compound is in second normal form iff it is in 1NF and there are no direct or transitive SD among its components.

Figure 8 shows a fragment of an algorithm that is in 1NF but not in 2NF. The algorithm is in 1NF even if we regard individual expressions as basic building blocks. However, the combination of f and g into a larger compound will violate 2NF because g and f posses a trivial SD.

Here, we used a textual notation to illustrate 2NF. A state diagram example violating 2NF is given in figure 3 with the SDs $X \overset{s}{\to} Y$ and $Y \overset{s}{\to} X$ violating 2NF.

VI. Related Work

Besides the references already given to normal forms in RDBT [7], [8], [9], software metrics [5] and software evolution, e. g. [1], [2], a few further works should not be left unmentioned in the context of this paper.

C. A. R. Hoare discusses *normal forms* of algebras and programming languages in [20] and distinguishes finite and infinite normal forms. A normal form of a programming language is defined by a set of rules posing restrictions on the available operators and the order of their application. The normal form must of course still be expressive enough so that any word of the non-restricted language can be transformed into the restricted form. For example, requiring that `while`-loops may not be used in a restricted version of Java defines a (rather dull) normal form, already. At an early stage of my considerations C. A. R. Hoare told me that normal forms by itself are not of much interest but the intentions behind the rules defining a certain normal form are. One goal behind normal forms in his book is testing two arbitrary programs for equality. This motivation differs from the work presented in this paper. Our goal is to define rules that lead to normal forms that maximize the maintainability of computer programs just like normal forms in RDBT do for data.

Because of its strong influence on the answer to the question whether to evolve, restructure, or replace a legacy system the assessment of the ability of a system to adapt to future changes is a major concern in the Software Re-engineering Assessment Handbook (SRAH) [15]. SRAH analyzes the evolvability of the code in a structured technical assessment by regarding amongst others: metrics, programming languages used, documentation, and age of data files. Though this approach has already proved its usefulness it could be improved with the addition of sound maintainability criteria that allow for a systematic in-depth analysis of the code. Code normal form rules like the ones presented in this paper could fill this gap.

The work presented is clearly closely related with software metrics [21], [22], [4], [23], [24]. Metrics analyzing various properties of programs have been developed in large numbers for different families of languages. It is well-known that metrics have their own anomalies [25]. For example, an increased cyclomatic complexity does not necessarily mean that the maintainability of the code has decreased. Vice versa, it is also commonly known that not every actual code decay becomes manifested in metrics. Hence, metrics are mathematically spoken neither sufficient nor necessary criteria. However, they are valuable indicators directing the attention of a software analyst to conspicuous locations within in the code for further manual investigation.

The main reason for the weakness of metrics is their syntactical nature. Even complicated rearrangements and mixtures of various metrics into higher level metrics such as in [26] rest solely on textual properties of the code. But, as we have argued above (see section IV) the main drivers for code modification troubles are semantic dependencies which can not be detected by solely regarding the text; just as it is impossible to find FDs in data bases by viewing attributes and tables, only. The key to detect shortcomings is semantic knowledge. However, it is to assume that metrics somehow correlate with the number of semantic shortcomings which make metrics useful albeit their imperfection.

VII. Conclusion and Future Work

Today, it is widely accepted that most successful software systems are inevitably subject to change over time. However, there is still a lack of knowledge about the factors influencing the productivity and quality of software modifications as well as their possible improvement.

This paper presented a first step towards a systematic definition of the maintainability of code. The main idea was to map the concept of normal forms from relational data base theory to code. It was shown how the update, delete, and insert anomalies known from data can be transfered to code and it is claimed that these anomalies are the source for various difficulties when changing code. This paper furthermore mapped sets and relations to atomic and composite algorithmic units and contributed a definition for semantic dependencies (SD) which form the foundation for code normal form criteria. These concepts were then introduce to state a first and a second normal form criteria for code that aim at separation of concerns and the elimination of redundancies via direct or transitive SDs. It is claimed that these concepts can be used for both assessing the maintainability of existing code as well as the systematic design of new maintainable code.

Since this is a completely novel approach it naturally leaves numerous open questions for future work from basic research to empirical studies of the concept. Clearly, the definition of SDs has to be further refined and additional normal form rules must be found. Besides this, it is not yet clear whether a complete elimination of code change anomalies is even possible. In contrast to data, it might be possible that code can not be completely freed from change anomalies. While this would not put the approach itself in question it would modify the goal to the minimization of unwanted change effects which could in turn result in different normal form criteria.

Albeit these difficulties, the concept described in this paper provides a systematic foundation for a better understanding of code modification phenomena and a formal framework for the development of novel methods for assessing and improving the maintainability of code.

References

[1] D. L. Parnas, "Software aging," in *Int. Conf. on Software Engineering (ICSE)*. Sorrento, Italy: IEEE, 16–21 May 1994, pp. 279–87.

[2] M. Lehman, "Software evolution threat and challenge," Professorial and Jubilee Lecture, Oct. 2003, 9th Int. Stevens Award, ICSM 2003.

[3] D. Tran-Cao, G. Levesque, and A. Abran, "Towards an effective measurement of complexity," in *Int. Conf. on Software Maintenance (ICSM)*. Montreal, Canada: IEEE Computer Society, Oct. 2002, pp. 370 – 376.

[4] B. Henderson-Sellers, *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.

[5] H. Zuse, *Software Complexity, Measures and Methods*. Walter De Gruyter, 1990.

[6] S. A. Österreich, "sem – qualitätsbewertung von software," 1988, version 1.0.

[7] E. F. Codd, "Further normalization of the data base relational model," *IBM Research Report, San Jose, California*, vol. RJ909, 1971.

[8] P. A. Bernstein and N. Goodman, "What does boyce-codd normal form do?" in *6th Int. Conf. on Very Large Data Bases*. Montreal: IEEE Computer Society, Oct. 1980, pp. 245–259.

[9] C. H. LeDoux and D. S. P. Jr., "Reflections on boyce-codd normal form," in *8th Int. Conf. on Very Large Data Bases*. Mexico City: Morgan Kaufmann, Sept. 1982, pp. 131–141.

[10] *The Java Language Specification*, 1st ed., Sun Microsystems, Mountain View, CA, Oct. 1995.

[11] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*. Reading: Addison Wesley, 1985.

[12] M. Fowler, *UML Distilled: A Brief Guide to the Standard Modeling Object Language*, 3rd ed., ser. Object Technology Series. Addison-Wesley, Sept. 2003.

[13] V. T. Rajlich and K. H. Bennett, "A staged model for the software life cycle," *IEEE Computer*, pp. 2–8, July 2000.

[14] W. M. N. W. Kadir and P. Loucopoulos, "Relating evolving business rules to software design," in *Proc. of the Intern. Conf. on Software Engineering and Practice SERP'03*, B. Al-Ani, H. R. Arabnia, and Y. Mun, Eds., vol. 1. Las Vegas, NV: CSREA Press, June 2003, pp. 129–134.

[15] STSC, "Software Reengineering Assessment Handbook v3.0," STSC, U.S. Department of Defense, Tech. Rep., Mar. 1997.

[16] B. P. Lientz, P. Bennet, E. B. Swanson, and E. Burton, *Software Maitenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Reading: Addison Wesley, 1980.

[17] T. M. Pigoski, *Practical Software Maintenance*. Wiley Computer Publishing, 1996.

[18] V. Rajlich and P. Gosavi, "A case study of unanticipated incremental change," in *International Conference on Software Maintenance*. Montreal, CA: IEEE, Oct. 2002.

[19] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, tools*. Addison Wesley, 1986.

[20] C. A. R. Hoare and H. Jifeng, *Unifying Theories of Programming*. Prentice Hall, 1998.

[21] L. H. Rosenberg and L. Hyatt, "Applying and interperting object oriented metrics," in *Software Technology Conference*, Utah, Apr. 1998.

[22] G. Kohler, H. Rust, and F. Simon, "Assessment of large object-oriented software systems: A metrics based process," in *Object-Oriented Technology (ECOOP'98 Workshop Reader)*, S. Demeyer and J. Bosch, Eds., vol. 1543. Springer-Verlag, 1998, pp. 250–251.

[23] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, June 1994.

[24] X. S. Zikouli, "Object-oriented metrics a survey," 2000.

[25] G. J. Myers, "An extension to the cyclomatic measure of program complexity," *ACM SIGPLAN Notices*, vol. 12, no. 10, pp. 61–64, Oct. 1977.

[26] C. M. S. E. Institute, "Maintainability index technique for measuring program maintainability," World Wide Web, Jan. 2004.