2010 IEEE International Conference on Software Science, Technology & Engineering

Abstraction Level Hierarchy: the Model and its Significance for Software Engineering

Hanania T. Salzer
School of Education, Tel-Aviv University
Tel-Aviv, Israel
hanania.salzer@013net.net

***Abstract*—An abstraction level hierarchy models abstraction-concretization relationships between different types of specifications. The proposed model combines into a single, continuous partially ordered set, all types of specifications, such as requirement specifications, design specifications and even the program code. Information theoretic definitions were developed for the concepts of abstraction, concretization and abstraction level. The paper demonstrates how the model can be used to reason about software engineering issues. The purpose of this conceptual paper is to propose an underlying model for software engineering research.**

Keywords-Abstraction; abstraction level hierarchy; atomic requirement specification; design; entropy; requirements; requirement specification

I. INTRODUCTION

Longstanding observation obtained during field work in the software industry suggested that certain outcomes, useful to software development, result from the use of atomic requirement specifications (ATRSs) as opposed to the use of non-atomic requirement specifications [1]. However, even if the hypothesized properties of using ATRS were accepted, they would be of little significance before an underlying formal model would be made available.

A suitable model had to show not only what it meant for the specification of a requirement to be atomic, but also it had to show what it meant to be atomic at a certain abstraction level. This last requirement is because atomicity is of meaning only when associated with an abstraction level. The challenge is, therefore, to define the notion of abstraction level, which in turn requires the definition of abstraction in the context of software design.

A. *Atomic Requirement Specification*

The specification of a system, or any of its components, is a description of its interface [2]. This implies that specifying the internal design of a system or a component involves listing its components or subcomponents, and specifying each one. This insight establishes a recursive relationship between components along the hierarchy of a system's structure: the specifications of a component residing at a certain level of the hierarchy is, at the very same time, part of the higher level component's internal design. Furthermore, "in software development, decomposition *is* implementation" [3]. In other words, every specification statement is the specification of a requirement relative to a component at some level, and in the same time it is also an internal design specification of a higher abstraction level [4], [5], [6].

Well-formed specifications are abstract, unambiguous, traceable and validatable [7]. ATRSs are well-formed specifications that, in addition, are also the result of splitting complex specifications into elementary, or indivisible, specifications [1].

The motivation for atomization of requirement specifications includes the difficulty to assign contractual liability to non-atomic specifications [8] and the danger of overlooking information [1]. The term *atomic requirement* is used to denote simple specifications in contrast to more complex ones [9], [10] and requirements expressed in a single sentence with one "shall", but without excluding multiple logical predicates within it [11]. Atomicity is used in [12] as a dimension for individual specifications, but without defining the dimension. Harn *et al*. [13] give examples of "atomic issues", but not all of them are indivisible. Indivisibility is included as the criterion for atomicity of a specification in [14] and [15].

B. *Abstraction*

The concept of abstraction is well defined in dictionaries and encyclopedias and also in the Software Engineering literature [16] although not formally. The significance of the skill of abstraction for software engineering has been discussed by [18]. The definition of abstraction level for individual specifications received little attention.

The most common use of abstraction in the software related refereed publications seems to be concerned with the design of software, such as abstract data types and object oriented programming, and considerably less to the abstraction of requirements. Abstraction is used by [17] as a noun to describe primary conceptual elements within specifications, and not as an action on a specification or as a relation between specifications. The amount of information in an abstraction level is described by [19] as follows "the quantity of information in a model varies with the LoA [Level of Abstraction]: a lower LoA, of greater resolution or finer granularity, produces a model that contains more information than a model produced at a higher, or more abstract, LoA." Yet, no account is given for why is that so.

978-0-7695-4061-0/10 $26.00 © 2010 IEEE
DOI 10.1109/SwSTE.2010.11

IEEE computer society

II. THE MODEL OF THE ABSTRACTION LEVEL HIERARCHY

The model has two underlying components: the specification of a requirement and the parent-child link that exists between some pairs of specifications.

A. First Component: Requirement Specification

The first underlying component of the Model is the specification of requirements. The term *requirement* has a variety of definitions, such as: “a condition or capability needed by a user to solve a problem or achieve an objective” [7]. Well-formed requirement specifications are abstract, unambiguous, traceable and validatable [ibid.]. An atomic requirement specification (ATRS) is defined as a well-formed specification of a requirement that is not possible to subdivide into more elementary specifications at the abstraction level at which it is being considered [20]. The term *abstraction level* will be brought into context only in the following section.

A requirement specification consists of a condition, sometimes implied, and a corresponding operation. An ATRS is a predicate with an indivisible condition, that is, no “OR”-s, and an indivisible operation, that is, no “AND”-s. A practical technique to check a requirement specification for indivisibility is by a mental experiment; if the tested object, either software or document text, can come out of a test against a specification with the conclusion that it has partially passed the test then the specification is not atomic [1].

B. Second Component: Rationale-Concretization Link

1) The Concept of Rationale: For every specification stated by a stakeholder, such as R1 in Fig 1, we could ask the stakeholder “Why?” [21], [22], [23] because she expects us to develop not exactly what she asks for, as much as what she really needs [24]. Then the stakeholder may or may not be able to tell why she has that requirement.

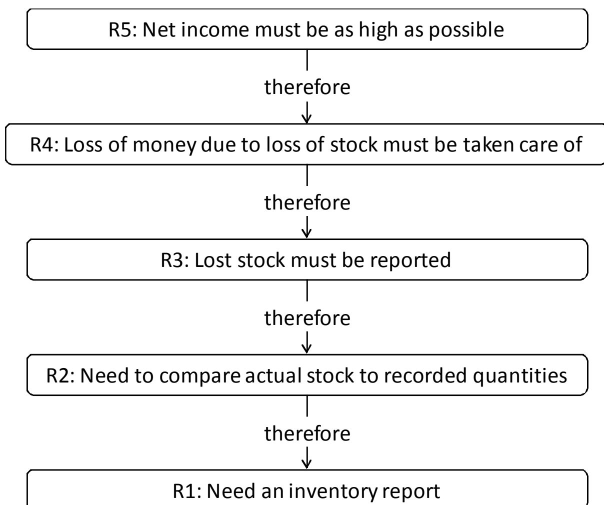

Figure 1. Rationale-Concretization links between specifications

As an arbitrary convention, a graphical representation can portray specifications as nodes, and can depict the links as edges pointing from the parent specification, or rationale, to the child specification, or concretization.

A specification may have any number of direct parents, or none. All direct parents of a specification, combined, are the rationales for that specification. A specification may have any number of children, including none. The combination of the children concretizes their parents. Thus a real life case is modeled by a graph far more complex than the one in Figure 1.

2) The Concepts of Abstraction and Concretization: R2 in Figure 1 is more abstract than R1 since R2 could be achieved not only with an inventory report, but also with a barcode reader. The report (R1) and the barcode reader are alternative concretizations of R2. Usually, a parent specification may be concretized in more than one, alternative way. By the time a system will be developed, one of the alternatives will be selected and further detailed. For this discussion, the relevant question is not “which one was implemented?” but “why specifically this one was preferred, and not the other one?”

According to this model, the reason that the designer selected R1 must be that she has had at least one more requirement, in addition to R2. In the presence of an additional requirement, such as “R7: We hate barcode readers”, the designer had had to select the report (R1), and had had to drop the barcode option, because only the former fulfilled both rationales. In other cases we must assume that the selection was affected by at least one rationale that was not explicitly specified, hence it is called *implicit rationale*.

3) The Concept of Abstraction Level: All specifications, by their very nature, carry information that prescribes some limitation on the freedom of choice from the set of different, alternative implementations of the system. The set of all systems that could implement a specification is called here the specification’s implementation space.

A specification carries some information about the appropriate, alternative designs of the system. The lower a specification is in the abstraction-level path, the more specific it is. The more concrete a specification is – the more information it carries about the system.

An abstraction level is defined indirectly, through a set of definitions of the inclusion of specifications in abstraction levels:

- Two specifications are said to be at the same abstraction level if, and only if, they have a common direct parent specification or a common direct child. In the former case, the two specifications are siblings.
- One specification is in a higher abstraction level then another specification if, and only if, the first specification is an ancestor of the other specification.
- One specification is in a lower abstraction level then another specification if, and only if, the first specification is a descendant of the other specification.

- When there is no ancestor-descendant relationship between two specifications, then the comparison of their abstraction levels is meaningless.

In the case of a rationale and one of its concretizations, R1→R3, it is conceivable to create another specification, say R2 such that R1→R2→R3. This mental exercise can be generalized into what seems to be a rule: for any two specifications, where one is the direct ancestor of the other, it is possible to insert between them a new specification. If indeed this is so, then the length of path between two specifications, a predecessor and a descendent, is of no quantitative meaning.

We define *natural abstraction level* as the location of a specification within an abstraction level hierarchy, when:

- all of its parents are its rationales, and
- all of its rationales are its ancestors, and
- all of its children are its concretizations, and
- all of its concretizations are its descendents.

III. Properties of the Model

The abstraction level hierarchy has a number of properties that can be delineated even before a formal account is established.

A. Stakeholders

Requirement specifications are stated by stakeholders who might, advertently or inadvertently, affect or be affected by the intended system. It is accepted by the trade that only clients, key users and the like define requirements [16]. The proposed model allows extending the concretization process below the specifications stated by clients, thus it allows including all stakeholders among those whose declarations, wishes and needs count as requirements.

Once all types of stakeholders, including designers and programmers, may concretize the existing requirement specifications, thus adding more specific requirements, the nature of the specifications of requirements becomes more versatile.

B. Abstraction Sequence and Concretization Sequence

Abstraction and concretization are two processes on the same specification, and they create sequences of specifications in opposite directions. Adding a rationale to an existing specification is an act of abstraction. Selecting one out of several alternative specifications that comply with the same rationales is an act of concretization. As stated above, detailed design specifications are concretizations of client requirements within the hierarchy of abstraction levels. Similarly, statements of program source code are concretizations of design specifications.

Every specification may be the rationale, or the "what", for a more concrete specification and in the very same time also the concretization, or the "how", of another, higher-level rationale [5], [25]. Therefore, all specifications, at all abstraction levels are the specifications of requirements [1], [3].

We define *executable source code* as a software artifact that is created directly by a human being and can be executed, usually after compilation, with the objective to exhibit the intended functionality. In contrast, executable machine code is not created directly by humans but by automatic processes such as the compilation of executable source code. Note that executable machine code is at a lower abstraction level than executable source code. For that to happen, the source code cannot be the sole rationale for the machine code. Other rationales, specified by humans, are the choice of software development environment, the choice of compiler arguments, the choice of operating system, and more. A formal specification is created directly by humans, and might be executable. If, in addition, it exhibits all of the intended functionality then, indeed, it is an executable source code.

The underlying properties of the abstraction level hierarchy do not predict any stopping point, or upper binding, in the sequence of successive abstraction. On the other hand, the sequence of successive concretization stops at the point of executable source code implementation. Because of this wide range, the model is not limited to requirements specified by clients or to any other arbitrary level of abstraction. On the contrary, it spans from the most abstract specifications to the most concrete specifications, namely the executable source code.

C. Arbitrary Abstraction Levels and Categories

Different authors advocate different numbers of abstraction levels, such as three [26], four [3], [27], five [28] and three to five [29]. Interestingly, the IEEE SWEBOK [30] and the SEI SWE-BOK [31] are mute about abstraction levels.

All of the above models have in common

- a similar underlying idea of abstraction levels
- a finite number of abstraction levels
- a very small number of abstraction levels, and
- abstraction levels that are arbitrary.

Of all the above properties, the models of fixed or arbitrary abstraction levels have only the first property in common also with the model presented by this paper.

Small, fixed number of abstraction levels is criticized by [32], for example on the ground that the connections between abstraction levels lack meaning. The arbitrary abstraction levels, which are suggested by the models of fixed abstraction levels type, were not derived from the needs of an abstraction-concretization process, but from the need to aggregate [3] specifications in categories. Categorization was expected to meet such needs as associating specifications with stages in the workflow, grouping specification by the different types of stakeholders, grouping specifications into different document types, organizing specifications to meet the standards in different software development methods or the capabilities of computer-aided software engineering tools. So, the models of fixed abstraction levels are motivated primarily by a need to categorize specifications; the perceived abstraction levels emerging from categories are only a byproduct.

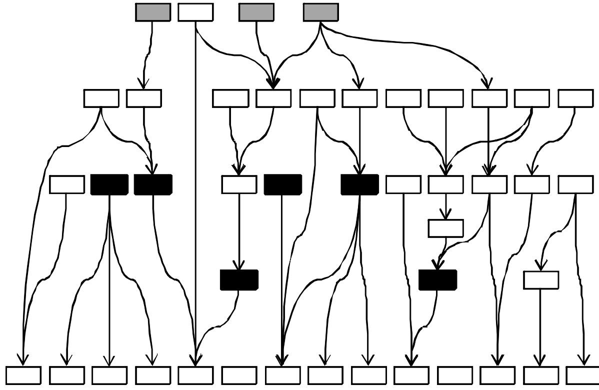

Figure 2. Aggregates (shading) and abstraction-concretization links (arrows) as two orthogonal properties of ATRSs (rectangles)

D. Comparison between Arbitrary and Continuous Abstraction Levels

Figure 2 illustrates the relationship between the hierarchy of requirement specifications and aggregates of requirement specifications in documents. Only two subsets, highlighted in gray and black, are documented. The figure shows that abstraction-concretization links and traditional aggregation can live together.

The arbitrary abstraction levels' approach has advantages, but also limitations in practical use. There is no doubt that some intermediate levels do exist in the mind of the designer, but a method of an arbitrarily limited number of abstraction levels results either in the loss of that information, or in different abstraction levels squeezed into and mixed up in a single document.

The Software Engineering industry attributes much importance to tracing back from specifications and from source code to their rationales [30], [33], [34], [35], [36], [37], [38]. Real path lengths from documented specifications back to their rationales may be larger than the limited number of arbitrary abstraction levels compelled on a development project. A hierarchy with an unlimited number of abstraction levels supports sequences of rationales with any length.

IV. A Formal Account of the Model

Of all specifications that are written by a human, the most concrete set of specifications is the executable source code. It is the hierarchy's lower binding. Although it is the endpoint of the design process, it offers a useful starting point for a formal analysis of the abstraction level hierarchy model.

A. The Quantity of Implementation Information in Specifications

Assume a small piece of source code picked out of a large system. It concretizes a specification statement, R20, picked out of the system's body of specifications, as in Figure 3.

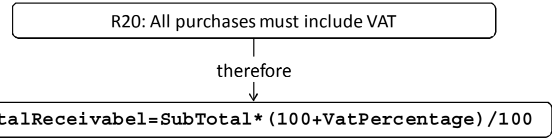

Figure 3. A piece of executable source code with a corresponding requirement specification

R20 carries some information that is necessary, but is not sufficient for implementation. The programmer must add to it from her knowledge and personal preferences: how to compute the VAT, what names to give to variables, and more. The line of code in the above example could have been written in several different ways. It is due only to some arbitrary decisions made by the programmer that of all alternatives this particular executable line of code was implemented. For a set of specifications we use the term *alternative implementation* for any one of the different, potential implementations that fulfill that set of specifications. Each alternative, has it been implemented, it would have successfully passed the tests against the said set of explicit specifications.

We define *implementation information* in a specification as the information in a specification that is available for predicting what executable source code will eventually implement that specification.

Looking at a specification, and considering its implementation space, we see that the potential manifestation of each one of the alternative implementations is a possible event. Whether we look at the executable source code written with a finite alphabet or at the executable machine code on a binary storage – these events are discrete. By their definition, all alternative implementations equally fulfill the specification. Therefore, if one is asked to predict, based strictly on the information in the specification, "what is the probability of the event that in the end a particular alternative implementation will be selected for actual implementation?" then the answer must be that all the events have the same probability. With no further information available, one can assume that this is a random event. It is also assumed that a real system is realized by software that was coded directly by human beings; hence the software has a finite size. Written with a finite alphabet, such software may have only a finite number of alternative implementations, thus making the events' probability finite too. This follows that the amount of implementation information available in any given specification is a case of a random variable of discrete events. Consequently, Shannon entropy [39], [40] can be used to model the information, or uncertainty, regarding to which of the alternative implementations will eventually be implemented. In an analogy to [39], the uncertainty in a specification may be regarded as the equivalence class of all translations of the same specification information into different, alternative implementations, thus reinforcing the current definition for implementation information in a specification.

Let n be the size, or the number of alternative implementations, of X, a specification's implementation space, then the entropy of that specification is $H(X)=log_2(n)$ bits. $H(X)$ is a measure of how little the specification's text

reveals about its eventual implementation. Thus entropy is a quantification of the implementation information, which was defined above qualitatively. The larger is *n*, that is the larger is the number of alternative implementations of a specification, the larger is its entropy, that is, the lower is that specification's capability to predict, or to provide information of what implementation will eventually prevail.

Suh [25] computes entropy of specifications at lower abstraction levels by the estimated probability that they will satisfy the specifications at the highest abstraction level. This paper makes no attempt counting the number of alternative implementations; hence, it cannot compute concrete values of entropy.

The implementation by an executable source code is a special set of specifications in regards to its entropy. Given that a certain line of code has been implemented, the probability that that line of code will be implemented is 1, and the entropy of that line of code is zero. Similarly, the entropy of any implementation of executable source code is zero.

B. Implementation Information in an Information Theoretical View

A specification can be viewed as an encoded message, and the set of alternative implementations can be viewed as its alphabet. The implementation, which is the executable source code, can be viewed as the decoded message. The message is encoded by means of the specification's utterance, which is usually non-formal. The decoded message is exactly one character of the alphabet. To reveal this single character, that is, to find out what is the executable source code, the message has to be decoded. Conversely, to abstract means to encode. This view allows reasoning about the implementation information in system specifications along the lines of Shannon entropy.

The text of a specification encodes some information about the program code that will eventually implement it. Encoding is non-algorithmic even in the case of formal specifications, except the program code itself. Decoding is the non-algorithmic, creative task of human designers.

C. Abstraction Levels and their Relationships

The implementation space of a specification must be a proper subset of the implementation space of each specification that is its rationale. It follows that the implementation space of a specification in the abstraction level of the concretization must be a proper subset of the intersection of all implementation spaces of all specifications in the abstraction level of rationales. This agrees with the intuitive idea that designers specify multiple specifications with the intention that the system will comply with all of them, as illustrated by Figure 4.

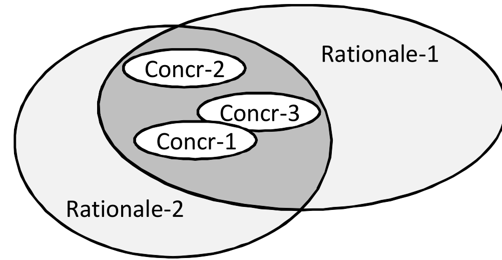

Figure 4. Reduction of implementation space with concretization

The intersection of the implementation spaces of all specifications in the abstraction level of the rationales contains less alternative implementations than is associated with any of the specifications in that abstraction level. Therefore, as one would expect, the entropy of this intersection is less than the entropy of each specification alone. We call the intersection of implementation spaces at the abstraction level of rationales the implementation space that is *bestowed* by the rationales to the concretization. Similarly, we call the entropy of the above intersection the entropy, or implementation information, bestowed by the rationales to the concretization.

Every specification in the abstraction level of the concretization must have its implementation space to be not only a subset of the bestowed implementation space, but it has to be smaller than that, otherwise no concretization has occurred. Since the implementation space of every specification in the abstraction level of concretization is smaller than the bestowed implementation space, it follows that the entropy in every concretizing specification is less than the entropy inherited from the abstraction level of rationales. Thus, concretization keeps decreasing the entropy.

V. PRACTICAL CONSEQUENCES OF THE ABSTRACTION LEVEL HIERARCHY

The abstraction level hierarchy model has a number of consequences that might be considered by practitioners and theorists.

A. Analysis and Design

We define *analysis* as a process that facilitates the identification of the design space and the set of possible alternative designs within it. We define *design* as the action of choosing specifications from among valid alternatives. Therefore, an analyst is also a designer, and vice versa.

Requirements engineers on the one extreme and programmers on the other extreme follow the same underlying principles of analysis and design activities. Both work on the same abstraction level hierarchy, but differ in taking responsibility for different regions of that hierarchy.

Designers, let alone programmers, add a huge amount of explicit specifications, many of which do not concretize explicit client requirements.

B. Ambiguous Specifications and Specifications of Non-functional Requirements

Requirements Engineering's current best practice condemns ambiguous specifications [41], [42], [43], [44],

and for a very good reason. The requirements engineering books tell us that we must replace the relics of raw, fuzzy specifications with less ambiguous ones, repeating the process until only the unambiguous specifications remain. We must document the latter in places such as a Software Requirement Specifications (SRS) document [41].

While a specification is ambiguous, from the point of view of an SRS, it still could be the most specific statement at the point of time when it emerged. In other words it is unambiguous at its natural abstraction level. Instead of replacing specifications, the abstraction level hierarchy model allows newer specifications to be added to the older specifications, thus maintaining a trace of rationales. It supports even the specifications that are not likely to be ever documented, such as a hidden agenda [36].

Vagueness and uncertainty emerge also while doing design and even during detailed design. The only really non-ambiguous construct is the executable source code.

C. *Specifications and Program Code*

The code written by a programmer is at the lowest abstraction level created by a human being during the development of a software system. From this point on, down the abstraction level in the direction of concretization, no more human concretization takes place, since a compiler or an interpreter takes over and creates an executable machine code.

Preprocessors, compilers, interpreters, linkers, database handlers, etc. implement a wealth of requirements. In a large project, only one or a few technical stakeholders specify which of these infrastructures to use for the system. The executable machine code concretizes these project-wide specifications together with the program code written by programmers.

Program code constitutes a very special case in the abstraction level hierarchy. No code is further concretized by human action, except starting off the compilation. Thus from the point of view of human creative action, program coding is the last step in the concretization process, and program code is the most concrete specification.

D. *Testing and Bugs*

The definition of testing relies on the definition of bug. The latter, in turn, relies on the definition of expected result. Therefore, they are discussed here in reverse order.

1) Expected Results: Before a scientist puts a hypothesis to the test, she predicts a-priori the outcome that the hypothesis and accepted theories predicts and the outcome that should be expected in the case that the hypothesis is false. By the experiment she compares the behavior of the set of real laws of nature with that of the set of accepted theories and her hypothesis. It is safe to say that the two sets are independent. This is not the case with software testing.

Oracle is a means that provides information about the correct, expected behavior of a component [45]. Software programmers have explicit and implicit information in the role of oracle: a subset of the abstraction level hierarchy of specifications and any implicit specifications maintained by the programmer herself. The software tester should have exactly the same explicit specifications. As a result, and in harsh contrast to scientific experiments, the oracle of the tester and the computation made by the program are dependent.

2) Bugs: Since the design space of a concretization must be contained within the design space of its rationales, we define bug as a contradiction between the information that exists in each one of two sets of specifications where one set is the ancestor of the other. The definition does not pretend to know which one of the two sets of specifications has the bug, and which one, if any, is correct. Furthermore, if a set of specifications in an abstraction level has an internal inconsistency, then the information bestowed by that abstraction level is null, which automatically renders it useless for being the source of information for any lower abstraction level.

Harwell *et al.* [4] define a specification's quality as the extent to which it reproduces in the mind of a reader the intellectual content that was in the mind of the writer, that is, how well it survives a handoff.

As we think up a specification it is in the form of a mental model. When we tell it to somebody, or write it down, the spoken and the written specifications are conceptual models. In the mind of the person who listens to or reads the conceptual model, a new mental model is being created. Both processes, from mental to conceptual model and from conceptual to mental model, are handoffs. Along the course of the system design process, specifications are handed off many times, alternating between mental and conceptual models.

Norman [46] describes mental models as incomplete, not accurate, and containing errors and contradictions; still, people will keep using a mental model even when they know that it is deficient. Therefore, it is reasonable to assume, that every time a model passes through a mental model, it emerges in a conceptual model after, potentially, attracting bugs.

Every act of concretization on a specification statement involves handoffs, making their number proportional to the number of links in an abstraction level hierarchy. Each handoff is a potential source of bug. This leads to the proposition that, after all, all bugs are specification bugs.

3) Tests: We define testing as the act of looking for information that exists in one set of specifications, called the oracle, but is absent from a second set of specifications when the oracle is at a higher abstraction level. A more practical definition says that a test tries to identify in the design space of the tested set of specifications a point that is outside the design space of the oracle, which is the set of specifications at the higher abstraction level. Note that the design space of the oracle should include the design space of the lower abstraction level, while the information in the oracle should be a subset of the information of the lower abstraction level.

The objects to be tested are software components as well as specification documents. The oracles are specification documents. Both items, the tested object and the oracle, are items in the same abstraction level hierarchy. Not only that but also the specifications that provide the oracle are always ancestors of the tested object.

Because oracles are ancestors of the tested objects, the bugs that were injected above the oracle, are included in the oracle, and were handed off, down the abstraction level hierarchy to the descended object under test. There is no way that the oracle will help exposing these bugs. The closer is the oracle to the tested object in terms of abstraction — the larger is the proportion of the handoffs that are common to both, hence the larger is the proportion of the bugs that the oracle cannot help discovering. On the other hand, the farther away is the oracle from the tested object in terms of abstraction — the less is the information that it carries about the tested object, and hence the weaker is its capability to act as an oracle.

Thus, the model shows that software testers must make a tough decision: how much information to trade for how much chance to catch bugs.

E. Source of Link

We used to think of the source of a requirement as an identification of the stakeholder who specified the requirement. We would assume that the source is an attribute of the specification.

With the proposed model, a stakeholder not only specifies that she has this-and-this requirement, but also she can point out what is her rationale for that specification. A specification may have multiple rationales, or it may have none. Different stakeholders may support the same link between a specification and its rationale, but for different reasons.

That being the case, the source of a requirement is not an attribute of a specification, but an attribute of a link between a specification and one of its rationales. A link may have multiple instances of the source attribute.

F. Lexicons and Information Hiding

The lexicon, or set of terms, used by the specifications in an abstraction level reflects information in that abstraction level. Because the information in an abstraction level is always less than the information in any abstraction level that is more concrete, an abstraction level cannot use information that will be a future contribution of a lower abstraction level. In practical terms, a specification must use terms only from the lexicons of the abstraction levels above it, and terms introduced by its own abstraction level, including itself.

The last statement sounds as tautology, but the following will show that this is not the case. The concretization of a specification, say R1, requires that the next abstraction level will introduce into its lexicon several new terms. Should the abstraction level of R1 itself refer to those terms, it would expand its own information to include the information of the next abstraction level, and by that making the next abstraction level redundant. Merging adjacent abstraction levels is possible, but then we could as well merge the whole abstraction level hierarchy into a single level, which is the executable source code. This is what hasty programmers do when they start coding before planning.

The restriction on the terms that specifications may use is an extension of information hiding from the description of modules [47] to the specifications at abstraction levels where modules, let alone their interfaces, have not yet been decided. Lexicon terms of an abstraction level are its *secret*, and they are hidden not only from higher abstraction levels but also from any abstraction level that is not on the same path. Information hiding is supported, therefore, by the abstraction level hierarchy not only vertically along paths of the hierarchy, but also sideways. The model expands the traditional notion of information hiding beyond the implementation, covering also the early specifications.

VI. Discussion

A model is proposed, which is formally based on a single relationship attribute, the abstraction-concretization relation that exists between pairs of specifications, providing a partial order over sets of specifications. It is visualized by a directed graph. The prevalent activity of design is concretization. Still, it was felt that the notion of abstraction already established a foothold in this context, and therefore the model is named the abstraction level hierarchy.

Partial order relates the specifications vertically. But still there was the need to define the inclusion of several specifications in the same abstraction level, by means of a horizontal relationship. Without that, we could talk only about the abstraction level of individual specifications, and were not able to speak of a set of specifications that comprised an abstraction level. The inclusion of two specifications in each other's abstraction level depends on the temporal nature of their direct parent specifications and their direct child specifications. These relations are circumstantial, because they may change as the design of a system proceeds.

Current Software Engineering practice views the rationale and the specification as two different entities, while in the abstraction level hierarchy model there is only one entity for both. On the other hand, traceability is separated into two, the abstraction-concretization link among specifications, and the source for each link, such as people and documents.

Testing is sustained by the abstraction level hierarchy beyond the support of requirements-based testing. The model shows that there is a vicious tradeoff between two factors, the ability to predict expected results and the ability to expose bugs.

Harel [48] advances the prospect of Visual Formalisms for specifications. Its visual facet makes specifications communicative for humans. Its formal facet makes specifications communicative with machines, but not only for the extent of consistency checking but actually to execute on machines. Harel's vision includes this: "We will … formulate and reformulate our conceptions as a series of increasingly more comprehensive models …". Still, the act of formulation and reformulation remains the very same non-algorithmic task as with non-formal and non-visual

specifications. Design is non-algorithmic even when its work product is a formal construct.

The sum of information entropy in a closed system is assumed to be constant [49]. However, a set of emerging system specifications is an open system; information enters it by the creative process of design, hence its entropy progressively decreases until, as it is fully decoded into the program code, it reaches zero.

The present work does not suggest that practitioners should document all rationales for all specifications in a mammoth abstraction level hierarchy. Such an attempt would be impossible and of little practical value. Neither does it suggest a new methodology; it can be applied to the current methodologies. The aim of this paper is to propose a necessary, underlying model for the toolbox of software engineering researchers.

ACKNOWLEDGMENT

This work was supported in part by a grant from the Constantiner family.

REFERENCES

[1] Salzer, H. (1999). “ATRs (Atomic Requirements) Used throughout development lifecycle.” 12th International Software Quality Week (QW99), 1, (6S1), San Jose, CA.

[2] Britton, K. H., & Parnas, D. L. (1981). “A-7E software module guide.” (NRL Memorandum Report 4702): Naval Research Laboratory (NRL).

[3] Wieringa, R. J. (1996). Requirements Engineering: Frameworks for Understanding. John Wiley & Sons Inc.

[4] Harwell, R., Aslaksen, E., Hooks, I., Mengot, R., & Ptack, K. (1993). “What is a requirement?” Proceedings of the Third International Symposium of the NCOSE.

[5] Kilov, H., & Ross, J. (1994). Information Modeling. Prentice-Hall.

[6] Ghezzi, C., Jazayeri, M., & Mandrioli, M. (2003). Fundamentals of Software Engineering, 2nd Ed., Upper Saddle River, New Jersey: Prentice Hall.

[7] IEEE. (1998). IEEE Std 1233: “Guide for Developing System Requirements Specifications” in IEEE Standards, Software Engineering, Volume One, Customer and Terminology Standards (IEEE, Computer Society). NY, USA: The IEEE, Inc., IEEE-SA Standards Board

[8] Hunt, L. B. (1997). “Getting the requirements right–a professional approach.” Eighth International Workshop on Software Technology and Engineering Practice (STEP '97) (including CASE '97), 464-472. IEEE Computer Society.

[9] Bolton, D., Jones, S., Till, D., Furber, D., & Green, S. (1992). “Know-ledge-based support for requirements elicitation: A progress review.” Technical Report TCU/CS/1992/23 and GMARC Project Report R44. City University.

[10] Sistla, P., Yu, C. T., & Venkatasubrahmanian, R. (1997). “Similarity based retrieval of videos.” Proceedings of IEEE ICDE 181-190. Birmingham, UK.

[11] Mannion, M., Keepence, B., & Harper, D. (1998). “Using viewpoints to define domain requirements.” IEEE Software, 95-102.

[12] Maiden, N., Minocha, S., Manning, K., & Ryan, M. (1997). “A Software Tool and Method for Scenario Generation and Use” in Proceedings of the Third International Workshop on Requirements Engineering: Foundations of Software Quality (REFSQ'97), 223-238.

[13] Harn, M., Berzins, V., Kemple, W., & Luqi (1999). “Evolution of C4I systems.” Monterey, CA 93943: Computer Science Department, Naval Postgraduate School.

[14] Loconsole, A., & Börstler, J. (2003). “Theoretical validation and case study of requirements management measures.” (Report UMINF 03.02), Sweden: Umea Universitet, Faculty of Science and Technology, Dep. of CS.

[15] Ozkaya, I., & Akin, Ö. (2007). “Tool support for computer-aided requirement traceability in architectural design: the case of DesignTrack.” Automation in Construction, 16, 674-684.

[16] IEEE. (1990). IEEE Std 610.12-1990: “Standard Glossary of Software Engineering Terminology” in IEEE Standards, Software Engineering, Volume One, Customer and Terminology Standards (IEEE, Computer Society). NY, USA: The IEEE, Inc., IEEE-SA Standards Board.

[17] Goldin, L., & Finkelstein, A. (2006). “Abstraction-based requirements management.” ROA '06: Proceedings of the 2006 International Workshop on Role of Abstraction in Software Engineering (Shanghai, China) 3-9. New York, NY, USA: ACM.

[18] Kramer, J. (2007). “Is abstraction the key to computing?” Communica-tions of the ACM, 50(4), 36-42.

[19] Floridi, L. (2008). “The method of levels of abstraction.” Minds & Machines, 18, 303-329.

[20] Salzer, H., & Levin, I. (2004). “Atomic requirements in teaching logic control implementation.” International Journal of Engineering Education, 20(1), 46-51.

[21] Van Lamsweerde, A., & Willemet, L. (1998). “Inferring declarative requirements specifications from operational scenarios.” IEEE Transactions on Software Engineering, 24(12), 1089–1114.

[22] Van Lamsweerde, A. (2001). “Goal-oriented requirements engineering: a guided tour.” Fifth IEEE International Symposium on Requirements Engineering (RE'01), 249-263.

[23] Leveson, N. G. (2000). “Intent specifications: an approach to building human-centered specifications.” IEEE Transactions on Software Engineering, 26(1), 15–35.

[24] Beizer, B. (1984). Software System Testing and Quality Assurance. NY, USA: Van Nostrand Reinhold electrical/computer science and engineering series.

[25] Suh, N. P. (1998). “Axiomatic design theory for systems.” Research in Engineering Design, 10, 189-209.

[26] Eden, A. H., Kazman, R., & Hirshfeld, Y. (2006). “Abstraction classes in software design.” IEEE Software, 153(4), 163-182.

[27] Wainer, G. A., Daicz, S., de Simoni, L. F., & Wassermann, D. (2001). “Using the Alfa-1 simulated processor for educational purposes.” ACM Journal of Educational Resources in Computing, 1(4), 111-151.

[28] Ye, N., & Salvendy, G. (1996). “Expert-novice knowledge of computer programming at different levels of abstraction.” Ergonomics, 39(3), 461-481.

[29] Gorschek, T., & Wohlin, C. (2006). “Requirements abstraction model.” Requirements Engineering, 11(1), 79-101.

[30] Abran, A., & Moore, J. W. (executive editors). (2004). Guide to the Software Engineering Body of Knowledge (SWEBOK). IEEE Computer Society.

[31] Hilburn, T. B., Hirmanpour, I., Khajenoori, S., Turner, R., & Qasem, A. (1999). A Software Engineering Body of Knowledge Version 1.0. (Technical Report CMU/SEI-99-TR-004 ESC-TR-99-004), Carnegie Mellon University, Software Engineering Institute.

[32] Lind, M. (1999). “Making Sense of the Abstraction Hierarchy.” CSAPC99. Villeneuve dAscq, France.

[33] Leveson, N. G.. (2000). “Intent Specifications: An Approach to Build-ing Human-Centered Specifications.” IEEE Transactions on Software Engineering, 26(1), 15-35.

[34] Cleland-Huang, J., Settimi, R., Duan, C., & Zou, X. (2005). “Utilizing Supporting Evidence to Improve Dynamic Requirements Traceability.” Proceedings of the 2005 13th IEEE International Conference on Requirements Engineering (RE05). IEEE Computer Society.

[35] Ibrahim, S., Munro, M., & Deraman, A. (2005). "A Requirements Traceability to Support Change Impact Analysis." Asian Journal of Information Technology, 4(5), 345-355.

[36] Dutoit, A. H., McCall, R., Mistrik, I., & Paech, B. (Eds.). (2006). Rationale Management in Software Engineering. Berlin Heidelberg: Springer-Verlag.

[37] Tang, A., Jin, Y., & Han, J. (2007). "A rationale-based architecture model for design traceability and reasoning." The Journal of Systems and Software, 80, 918-934.

[38] Fogarty, K., & Austin, M. (2008). "System Modeling and Traceability Applications of the Higraph Formalism." Systems Engineering, electronic publication, DOI: 10.1002/sys.20113.

[39] Shannon, C. E. (1953). "The lattice theory of information." Transactions of the IRE professional Group on Information Theory, 1(1), 105-107.

[40] MacKay, D. J. (2005). Information Theory, Inference, and Learning Algorithms. Cambridge University Press.

[41] IEEE. (1998). IEEE Std 830-1998: "IEEE Recommended Practice for Software Requirements Specifications" in IEEE Standards, Software Engineering, Volume One, Customer and Terminology Standards (IEEE, Computer Society). NY, USA: The IEEE, Inc., IEEE-SA Standards Board.

[42] Robertson, J., & Robertson, S. (Template). (2000). Volere Requirements Specification Template (Edition 13). Atlantic Systems Guild.

[43] Berry, D. M., Kamsties, E., & Krieger, M. M. (2003). "From Contract Drafting to Software Specification: Linguistic Sources of Ambiguity, A Handbook." (Technical report, version 1.0), University of Waterloo: Canada.

[44] Wiegers, K. E. (2003). Software Requirements (2nd Edition). Redmond, Washington, USA: Microsoft Press.

[45] Beizer, B. (1990). Software Testing Techniques, 2nd edition. London: International Thomson Computer Press.

[46] Norman, D. (1983). Some Observations on Mental Models. In Gentner, D.; Stevens, Albert, Mental Models (7-14).

[47] Parnas, D. L. (1972). "On the Criteria to be used in decomposing Systems into Modules." Communications of the ACM, 15(12), 1053-1058.

[48] Harel, D. (1992). "Biting the Silver Bullet, Toward a Brighter Future for System Development." Computer, January, 8-20.

[49] Wang, Y. (2004). "On the Cognitive Informatics Foundations of Software Engineering." Proceedings of the Third IEEE International Conference on Cognitive Informatics (ICCI04): IEEE Computer Society.