Is there Evolution before Birth? Deterioration Effects of Formal Z Specifications

Andreas Bollin

Software Engineering and Soft Computing, AAU Klagenfurt, Austria
Andreas.Bollin@aau.at
http://www.aau.at/tewi/inf/isys/sesc

Abstract. Formal specifications are not an exception for aging. Furthermore, they stay valid resources only in the case when they have been kept up to date during all evolutionary changes taking place. As specifications are then not just written once, an interesting aspect is whether they do also deteriorate or not. In order to answer this question, this paper addresses the issues on various kinds of changes in the development of formal specifications and how they could be measured. For this, a set of semantic-based measures is introduced and then used in a longitudinal study, assessing the specification of the Web-Service Definition Language. By analyzing all 139 different revisions of it, it is shown that specifications can deteriorate and that it takes effort to keep them constantly at high quality. The results yield in a refined model of software evolution exemplifying these recurring changes.

1 Introduction

Would you step into a house when there is a sign saying "Enter at your own risk"? I assume not, at least if it is not unavoidable. Well, the situation is quite comparable to a lot of software systems around. Our standard software comes with license agreements stating that the author(s) of the software is (are) not responsible for any damage it might cause, and the same holds for a lot of our hardware drivers and many other applications around. Basically, we use them at our own risk.

I always ask myself: "Would it not be great to buy (and also to use) software that comes with a certificate of guarantee instead of an inept license agreement?" Of course, it would and it is possible as some companies demonstrate. It is the place where formal methods can add value to the development process. They enable refinement steps and bring in the advantages of assurance and reliable documentation.

The argument of quality is not just an academic one. Formal methods can be used in practice as companies using a formal software development process demonstrate [23]. Formal modeling is also not as inflexible as one might believe. Changing requirements and a growing demand in software involve more flexible processes and it is good to see that a combination of formal methods and the world of agile software development is possible [2]. This enables the necessary shorter development cycles, but, and this is the key issue, it also means to start thinking about evolution right from the beginning.

The questions that arise are simple: (a) Do our formal specifications really change or evolve, and (b) if this is the case, can we detect or even measure these changes? The objective of the paper is to answer these two questions. In a first step it demonstrates that formal specifications are not an exception for aging. Section 2 tries to make developers more receptive to this topic. And in the second step it demonstrates that there might be deterioration effects when formal specifications are undergoing constant changes. For this, Section 3 briefly introduces a set of measures that are suitable for assessing Z specifications, and Section 4 takes a closer look at 139 revisions of a large Z specification. Due to the lessons learned, a refined model of software evolution is suggested in Section 5. Finally, Section 6 summarizes the findings and again argues for a careful attention of the refined model of (specification) evolution.

2 Perfection or Decay

A formal specification describes what a system should do and as such it can be used to argue about the correctness of a candidate system. But a specification is not per se a "correct" mapping of the requirements. It needs time to create a first, useful version and, as there are affinities with traditional software development, this section starts with the model of software evolution. This model is then the basis for a – necessary – refinement, as is shown later in Section 5.

2.1 Back to the Roots

Let us start again with the analogy above: Why does one enter a house even without bothering about its safety? The answer is simple: normally, one trusts in the original design, the statics, the teams that built it and the officials that did the final checks. The trust stays the same when the house is going to be renovated, when the interior changes and when some walls are broken down (or new ones are erected). One naturally assumes that the old plans have been investigated and that structural engineers took a look at it. The same holds for our software systems. There is an overall design, there are teams that build and modify it and there are teams that test it before being sold. We trust in their professionalism. A change in requirements and needs then leads to a change in this software – it is undergoing a "renovation" process that we might call software evolution.

Bennet and Rajlich [1] introduced a staged model to describe this process in more details (see Fig. 1). Starting with the initial development and the first running version, evolutionary changes happen, leading to servicing phases and then, finally, to the phase-out and close-down versions of the software. In their article the authors also point out the important role of software change for both, the evolution and servicing phases. In fact, effort is necessary at every step of the phase to keep up with the quality standards and for keeping alive our trust in it.

Taking a closer look at our analogy of building/reconstructing a house it can be observed that there is also a chain (or network) of trust. The visitor (or owner) of the house counts on the construction company, they by themselves trust in the quality of the building materials they use, and the team that builds the house trusts in the architects

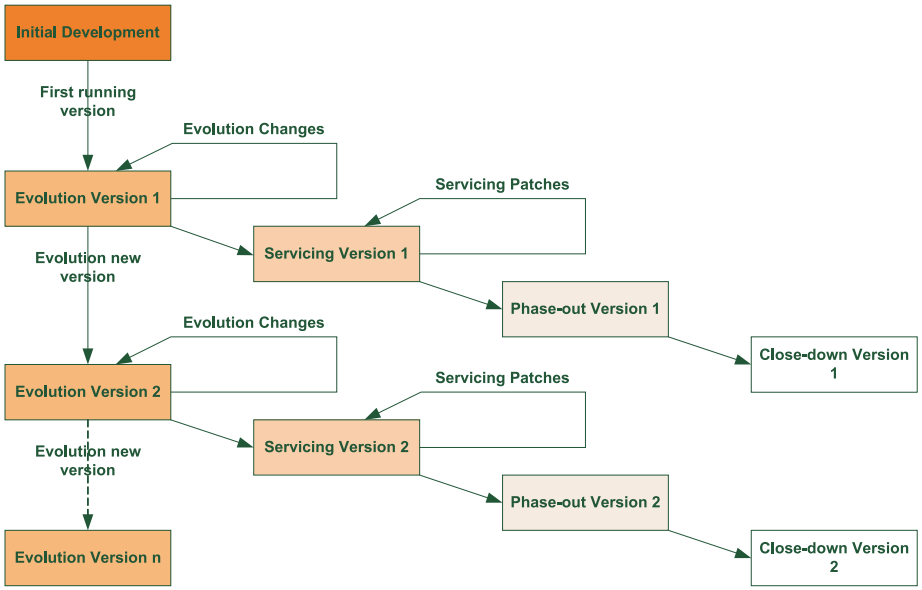

Fig. 1. The versioned staged model of Bennett and Rajlich [1]. Starting with the initial development and the first running version evolution is about to begin. The goal of the evolution phase is to adapt the software to ever changing user requirements or changes in the operating environment. When substantial changes are not possible anymore (at least not without damages to the system), then the servicing phase starts. Only small tactical changes are applied. When no more servicing is done, then the phase-out starts. Finally, with the close-down phase, the users are directed towards a new version.

(just to mention some of the links). When our evolving software is undergoing changes, then there is a similar chain of trust and dependencies.

This is now the place where formal methods come into play. To keep the trust, a change in the software has to be preceded by changes in the design documents and with it a change in the software specifications. One can also put it the other way round: when the architect does not update his or her plans, then future renovations are (hopefully) impeded.

2.2 The Role of Formal Design

Writing down requirements in a keen way is necessary, and the use of formality is not new in this respect. In their article Black et.al point out that formal methods have already been used by Ada Lovelace's and Charles Babbage's work on the analytical engine when they verified the formulas [2]. Since then several success stories of the use of formal methods have been published [8, 28, 23, 12, 11]. However, traditional formal design is commonly seen as something that is just happening at the beginning, and most of us are tempted to think about the development of just *one* formal model.

As the analogy above demonstrates, this viewpoint is probably not correct. When drawing up a plan, an architect does not only draw a single version. He or she starts with a first version, modifies it, and plays around with it. Several versions are assessed and, finally, discussed with the customer. The plans are, in addition to that, revised when the building is changed later on. The same holds for formal specifications. Their advantages not only lie in verification and validation considerations. They form the basis for incrementally pinning down the most important requirements (and eventually ongoing refinement steps). Only when kept up to date during evolutionary changes, they act as valid sources for comprehension activities. So, our formal specifications are (and should be) constantly changing during the software development phases. To think in terms of "write-once languages" (as already addressed in [16, p.243]) is for sure not appropriate.

During the last three decades there have been several advances in the field of formal software development. Specification languages were standardized and then also adapted to the world of object oriented programming. By the time new languages arise. Alloy is such an example that also allows for simulation and graphical feedback [13]. However, the main focus of the tools and languages around rests on support for writing the one and correct model (and on proving properties of it). Contrarily, in our working group we have been focusing on servicing and comprehension aspects instead [21, 17], and in the last years these efforts led to a concept location model and a tool for visualization, slicing and clustering of Z specifications [4]. The resulting tools and techniques will now be used in order to find out whether (and to which extent) formal specifications do change during evolutionary activities.

3 On the Search for Measures

Formal specifications (like program code) might age during modifications and it needs effort to antagonize it. The effects of a modification should be measured in order to steer the course of change. This means to assess the specification (among other documents) at every refinement step and to consider the effect on various parameters of the specification. Looking at size-based measures only (which can be calculated easily) is for our objectives not enough. When talking about various aspects of deterioration we are more interested in measuring effects on the specifications' *quality*!

3.1 Specification Measures

The majority of specification metrics used in projects belongs to the class of size/quantity based measures. Most popular is counting lines of specification text, which, apart from looking at the time needed to write the specification, was also used as *the* basis to monitor the often-cited CICS/ESA project of IBM [9]. Counting specific specification elements is possible, too. Vinter et. al propose to count the type and number of logical constructs in Z specifications [25]. By a small case-study they demonstrate that these measures might correlate with the complexity of the specification. However, up to now a quantitative assessment of the approach is missing. Nogueira et. al suggest to use two measures expressing the complexity of each operator in the system and to calculate

them by counting input and output data related to the operators [18]. Their experiences are based on a small case-study using *HOPE* as a specification language and Modula-2 for its implementation. Alternatively, Samson et. al suggest to count the number of equations in a module or per operation [24]. They observed that the number of equations required to define an operator is frequently equal to the cyclomatic complexity of code based on the specification.

Complexity considerations are relevant, but within the scope of this work quality measures are needed. Due to the declarative nature of the formal specifications under investigation, such measures (usually based on control- and data-flow) are, unfortunately, rare. The above mentioned approaches of Samson et. al or Nogueira et. al can be seen, if at all, just as possible approximations to quality considerations. But, there is one approach that could be used as a starting point. By looking at (and analyzing) the references to state variables in operations, Carrington et. al try to identify states of a module and to relate them to the top-level modular structure of an implementation [6]. With this, they are introducing the notion of cohesion within a module. They do not relate it to the quality of a specification, though, but the next section demonstrates that not so much is missing.

3.2 Slice-based Coupling and Cohesion Measures

As mentioned in the previous section, it is hard to find suitable quality measures for formal specifications. However, for programming languages there are several approaches around. Recently, Meyers and Binkley published an empirical study demonstrating the use of slice-based measures for assessing the quality of 63 C programs [15]. Their study is based on the following situation: In a system different relations between different components can be detected. These relations make up part of the class of semantic complexity. When taking the information flow within and between these components as quality indicators, then the dual measures of coupling and cohesion are quite descriptive when assessing the quality of a program.

A practical way to calculate the needed measures is to make use of slices. Weiser [26, 27] already introduced five slice based measures for cohesion, and three of them have later on been formally defined by Ott and Thuss [20]: Tightness, Coverage, and Overlap. Coupling, on the other hand, was defined as the number of local information flow entering and leaving a procedure, and Harman demonstrated in [10] that it can be calculated by the use of slices, too.

According to [15, 2:6-2:7], Tightness relates the number of statements common to all slices to the length of the module. It favors concise single thought modules where most of the statements are part of all the slices and thus affect all of the outputs. Coverage on the other hand relates the lengths of the slices to the length of the entire module. It favors large slices but does not require them to overlap and thus to indicate single thought modules. As an example, a module containing two independent slices would result in a value for Coverage of 0.5 but a value for Tightness of 0.0. Overlap measures how many statements are common to all the slices and relates the number to the size of all slices. The result is a measure that is not sensitive to changes in the size of the

Measure	*Definition*	*Description*
Tightness $\tau(\Psi,\psi)$	$\frac{\mid SP_{int}(\Psi,\psi)\mid}{\mid\psi\mid}$	Tightness τ measures the number of predicates included in every slice.
Coverage $Cov(\Psi,\psi)$	$\frac{1}{n}\sum_{i=1}^{n}\frac{\mid SP_i\mid}{\mid\psi\mid}$	Coverage compares the length of all possible specification slices SP_i ($SP_i \in SP(\Psi,\psi)$) to the length of ψ.
Overlap $O(\Psi,\psi)$	$\frac{1}{n}\sum_{i=1}^{n}\frac{\mid SP_{int}(\Psi,\psi)\mid}{\mid SP_i\mid}$	Overlap measures how many predicates are common to all n possible specification slices SP_i ($SP_i \in SP(\Psi,\psi)$).
Inter − Schema Flow $\digamma(\psi_s,\psi_d)$	$\frac{\mid(SU(\psi_d)\cap\psi_s)\mid}{\mid\psi_s\mid}$	Inter-Schema flow $\digamma$ measures the number of predicates of the slices in ψ_d that are in ψ_s.
Inter − Schema Coupling $C(\psi_s,\psi_d)$	$\frac{\digamma(\psi_s,\psi_d)\ \mid\psi_s\mid\ +\ \digamma(\psi_d,\psi_s)\ \mid\psi_d\mid}{\mid\psi_s\mid+\mid\psi_d\mid}$	Inter-Schema coupling C computes the normalized ratio of the flow in both directions.
Schema Coupling $\chi(\psi_i)$	$\frac{\sum_{j=1}^{n} C(\psi_i,\psi_j)\ \mid\psi_j\mid}{\sum_{j=1}^{n}\mid\psi_j\mid}$	Schema Coupling χ is the weighted measure of inter-schema coupling of ψ_i and all n other schemas.

Table 1. Coupling and cohesion-related measures for Z specifications as introduced in [5]. *SP* is the set representing all slices SP_i of a schema ψ in a Z specification Ψ. SP_{int} is the Slice Intersection, representing the set of all predicates that are part of all slices. *SU* represents the Slice Union of all the slices.

module, it is only related to the size of the (single) common thought in the module. Coupling between two modules is calculated by relating the inflow and outflow of a module (with respect to other modules in the program). Inflow and outflow are also calculated by making use of slices. Inter-procedural slices yield those statements of a module that are "outside", and, when examining these sets of statements mutually, their relation can be treated as "information flow". The exact semantics behind the measures (including the definitions and some impediments in calculating them) are explained in more details in the paper of Meyers and Binkley [15].

3.3 Specification Slicing

Slicing can be applied to formal specifications, too. The idea was first presented by Oda and Araki [19] and has later been formalized and extended by others [7, 3, 29]. The basic idea is to look for predicates that are part of pre-conditions and for predicates that are part of post-conditions. The general assumption is that (within the same scope) there is a "control" dependency between these predicates. "Data dependency", on the other hand, is defined as dependency between those predicates where data is potentially propagated between them. With this concept, slices can be calculated by looking at a (set of) predicates at first and then by including all other dependent predicates.

Metric Comparison (n=1123)								
Sig.			Pearson		Spearman		Kendall	
	Measure 1	*Measure 2*	*R*	*p*	*R*	*p*	*R*	*p*
Strong	*Tightness*	*Coverage*	0.830	.000	0.907	.000	0.780	.000
Moderate	*Tightness*	*Overlap*	0.809	.000	0.749	.000	0.623	.000
	Size (*LOS*)	*Coupling*	0.589	.000	0.686	.000	0.494	.000
	Size (*LOS*)	*Overlap*	-.557	.000	-.543	.000	-.415	.000
	Size (*LOS*)	*Tightness*	-.541	.000	-.551	.000	-.415	.000
	Coverage	*Overlap*	0.531	.000	0.566	.000	0.437	.000
Weak	*Coupling*	*Overlap*	-.343	.000	-.315	.000	-.239	.000
	Size (*LOS*)	*Coverage*	-.284	.000	-.447	.000	-.326	.000
	Coupling	*Tightness*	-.272	.000	-.262	.000	-.191	.000
	Coupling	*Coverage*	0.006	.829	-.102	.000	-.070	.000

Table 2. Pearson, Spearman, and Kendall Tau test values (including significance level *p*) for the correlation of size and slice-based Z specification measures. Values $\mid R \mid \in [0.8 - 1.0]$ in the mean are classified as strongly correlated, values $\mid R \mid \in [0.5 - 0.8)$ are classified as moderately correlated, and values $\mid R \mid \in [0.0 - 0.5)$ are treated as weakly correlated.

Recently, sliced-based coupling and cohesion measures have then been mapped to Z by taking the above definitions of Meyers and Binkley as initial points (see Table 1 for a summary). Based on the calculation of slice-profiles which are collections of all possible slices for a Z schema, the following measures have been assessed in [5]:

- Tightness, measuring the number of predicates included in every slice.
- Coverage, comparing the length of all possible slices to the length of the specification schema.
- Overlap, measuring how many predicates are common to all n possible specification slices.
- Coupling, expressing the weighted measure of inter-schema coupling (the normalized ratio of the inter-schema flow – so the number of predicates of a slice that lay outside the schema – in both directions).

In [5] it was shown that the measures are very sensitive to semantic changes in Z-schema predicates and that the changes of the values are comparable to their programming counterparts. This includes all types of operations on the specification, especially the addition, deletion or modification of predicates. The next and missing step was to look at a larger collection of sample specifications and assessing their expressiveness. The major objective was to find out which of the measures describe unique properties of a Z specification and which of them are just proxies for e.g. the lines of specification text count (LOS).

In the accompanying study more than $12,800$ lines of specification text in $1,123$ Z schemas have been analyzed and the relevant results of the analysis of the measures are summarized in Table 2. The table shows that three tests have been used: Pearson's linear correlation coefficient, Spearman's rank correlation coefficient, and Kendall's

Tau correlation coefficient. The objective was to find out whether each of the measures represents some unique characteristic of the specification or not.

The Pearson's correlation coefficient measures the degree of association between the variables, but it assumes normal distribution of the values. Though this test might not necessarily fail when the data is not normally distributed, the Pearson's test only looks for a linear correlation. It might indicate no correlation even if the data is correlated in a non-linear manner. As knowledge about the distribution of the data is missing, also the Spearman's rank correlation coefficients have been calculated. It is a non-parametric test of correlation and assesses how well a monotonic function describes the association between the variables. As an alternative to the Spearman's test, the Kendall's robust correlation coefficient was used as it ranks the data relatively and is able to identify partial correlations.

The head to head comparison of the measures in Table 2 shows that the slice-based measures are not only proxies for counting lines of specification text. In fact, most of the pairs do have a weak or moderate correlation only. So, besides the size of the specification, one can select Coverage, Overlap, and Coupling as descriptors for properties of the specification, but, e.g., skip Tightness as it has the highest values of correlation to most of the other measures.

Meyers and Binkley suggested another measure based on the sizes of the generated slices and called it "deterioration" [15]. This measure has also been mapped to Z in [5] and the basic idea goes back to a simple perception: the less trains of thoughts there are in one schema, the clearer and the sharper is the set of predicates.

When a schema deals with many things in parallel, a lot of (self-contained) predicates are to be covered. This has an influence on the set of slices that are to be generated. When there is only one "crisp" thought specified in the schema, then the slice intersections cover all the predicates. On the other hand, when there are different thoughts specified in it, then the intersection usually gets smaller (as each slice only regards dependent predicates). A progress towards a single thought should therefore appear as a convergence between the size of the schema and the size of its slice-intersection, a divergence could indicate some "deterioration" of the formal specification. This measure seems to be a good candidate for checking our assumption whether specifications do age qualitatively or not, and it defined as follows:

Definition 1 Deterioration. *Let Ψ be a Z specification, ψ_i one schema out of n schemas in Ψ, and $SP_{int}(\psi_i)$ its slice intersection. Then Deterioration ($\delta(\Psi)$) expresses the average module size in respect to the average size of the slice intersections SP_{int}. It is defined as follows:*

$$\delta(\Psi) = \frac{\sum_{i=1}^{n} \mid \psi_i \mid - \mid SP_{int}(\Psi, \psi_i) \mid}{n}$$

Please note that the term "deterioration" as introduced in this paper is neither positive nor negative and one single value of deterioration is of course not very expressive. It just tells about how crisp a schema is. It does not allow for a judgement about the quality of the schema itself. Of course, we could state that all values above a pre-defined

value x are to be treated as something unwanted, but it depends on the problem at hand whether we can (and should) allow for such schemas. In all, it merely makes sense to look at the differences in deterioration between two consecutive versions of the specification and thus to introduce the notion of Relative Deterioration. This measure can be defined in such a way that the relative deterioration is greater than zero when there is a convergence between schema size and slice intersection, and it is negative, when the shears between the sizes get bigger, indicating some probably unintentional deterioration. Relative Deterioration is defined as follows:

Definition 2 Relative Deterioration. *Let Ψ_{n-1} and Ψ_n be two consecutive versions of a Z specification Ψ. Then the relative deterioration ($\rho(\Psi_{n-1}, \Psi_n)$) with $n > 1$ is calculated as the relative difference between the deterioration of Ψ_{n-1} and Ψ_n. It is defined as follows:*

$$\rho(\Psi_n) = 1 - \frac{\delta(\Psi_n)}{\delta(\Psi_{n-1})}$$

4 Evaluation

With the set of measures at hand and the reasonable suspicion that specifications do age this paper is now taking a closer look at the development of a real-world specification and the effect of changes onto the measures introduced in Section 3.

4.1 Experimental Subject

One of the rare, big publicly available Z specifications is the specification of the Web Service Definition Language (*WSDL*) [22]. The document specifies the Web Services Description Language Version 2.0, which is an XML language for describing Web services. Besides natural language guidance, the specification defines the core language that is used to describe Web services based on an abstract model of what the service offers. Additionally, it specifies the conformance criteria for documents in this language. The reason for focusing on this specification is that, with 2004 onwards, a concurrent versioning system (*CVS*) has been used. *WSDL* 1.0 is not available in Z, but from November 2004 till the final release in 2007 139 versions have been checked in. The first revision is an adoption of *WSDL* 1.0, and then, successively, functionality has been added, modified, or deleted. The final revision contains 814 predicates (distributed over 1,413 lines of Z text).

This specification is now used so check whether, due to maintenance operations, there are drastic changes in the measures and whether deterioration can be detected or not. The strategy is simply to look at the changes (as documented in the code and in the *CVS* log files) and to compare them to the obtained values.

4.2 The Study

As a first step the *CVS* log was analyzed. This provided some insights to the types of changes that occurred on the way to the final release. Though there have been several

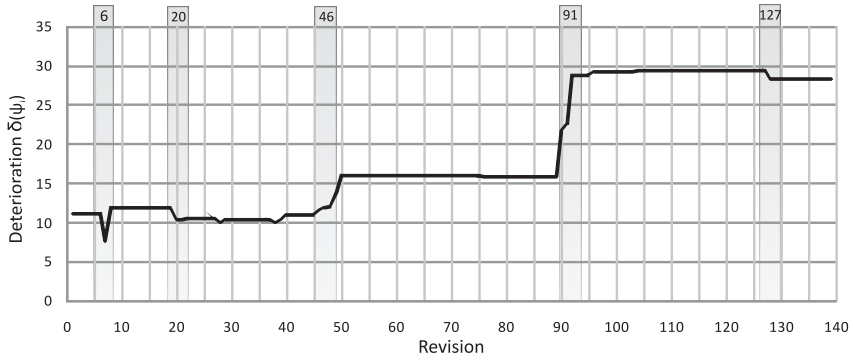

Fig. 2. Deterioration for the 139 revisions of the WSDL specification. When the value of deterioration increases, then more predicates (not closely related to each other) are introduced to schemas. This is not bad per se, but it is a hint towards a decrease in cohesion values.

changes influencing the events, the following sections and revisions are noticeable and are considered in more details:

- Up to Revision 1.005 there is a mapping of *WSDL* version 1.0 to Z. Only minor changes to the specification happen.
- Between revisions $1.005 - 1.007$ there are mostly structural enhancements. Finally, model extensions take place.
- At revisions 1.020*ff* there are several refactoring steps and noticeable extension.
- At revisions 1.045*ff* there are several smaller changes to some of the concepts.
- At revisions 1.090*ff* there are massive extensions to the model and new concepts are introduced.
- Between revisions $1.096 - 1.104$ the concepts in the model are simplified.
- At revisions 1.127*ff* there are change requests and, thereinafter, removing features leads to a structural refactoring.

Up to revision 1.092 the interventions consisted mainly of adding new concepts (in our case Z schemas) to the specification. After revision 1.095 there are solely change requests, leading to a refactoring of the specification. Really massive changes took place at revisions 1.046 and 1.091.

When taking another closer look at the *CVS* log and the code, a specific strategy for keeping the specification constantly at a high level of quality can be detected. The recurring steps of a change request were:

1. Refactoring of the actual version.
2. Adding, removing, or modifying a concept.
3. Update of the natural language documentation.

The interesting question is now whether our measures introduced in Section 3 are able to reflect these changes and whether the measures of deterioration are able to display these changes.

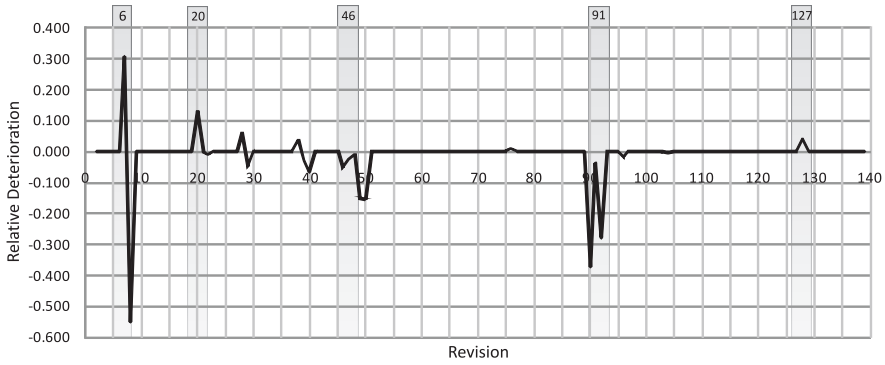

Fig. 3. The change in deterioration is better visible when looking at the relative deviation over the time. A positive value indicates an increase in cohesion, while a negative value indicates a decrease in the values of cohesion.

4.3 Results

At first let us take a closer look at the measure called deterioration. Fig. 2 presents the value for all 139 revisions in the *CVS*. This figure indicates that the specification remarkably changes at revisions 1.046 and 1.091. In fact, the *CVS* log also documents the changes.

As absolute values (as in Fig. 2) do not perfectly describe the influence of a change, the notion of relative deterioration has been introduced in Section 3. Fig. 3 presents the value of it for all 139 revisions. Positive values indicate that the difference between the schema sizes and their slice intersections is reduced; such a deviation is assumed to be positive in respect to deterioration as the slice intersection is a measure of how strong the predicates are interwoven in a schema. On the other hand, negative values indicate negative effects.

When taking again a look at Fig. 3 (especially between revisions 1.020 and 1.046), then the above mentioned strategy of change requests gets noticeably visible. A change is implemented by a structural improvement first (to be seen as a positive amplitude), and then it is followed by the introduction of the new concept, in most cases indicated by a negative amplitude in the diagram.

Let us now analyze the influence of a change onto the qualitative values of coupling and cohesion. By looking at Fig. 4, we see that the value for overlap decreases (on average) a bit. This indicates that, with time, the number of predicates, common to other slices, gets lower. Single Z schemas seem to deal with more independent thoughts. Refactoring these thoughts into separate schemas (which happened e.g. at revisions 1.020 and 1.127) helped a bit to improve the structure of the specification again.

The value of coverage follows more or less the fluctuation of overlap – but not at all revisions to the same extent. On the long run it definitely increases. Coverage tells us about how crisp a schema is, and in our case the developers of the specification did not manage to keep this property stable.

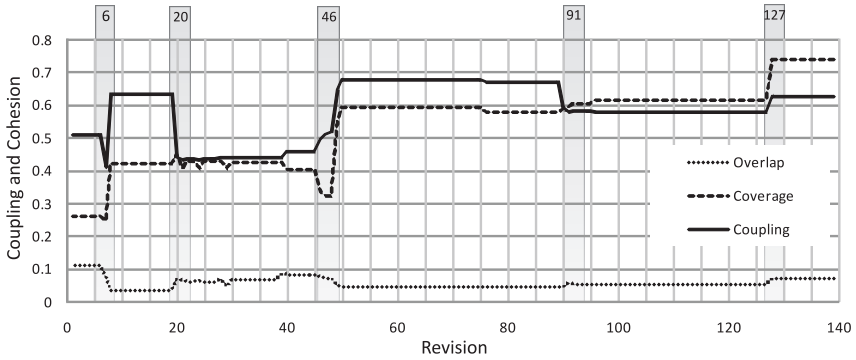

Fig. 4. The values of cohesion (expressed by the measures of overlap and coverage) and coupling for the 139 revisions of the WSDL specification. In most cases the values are subject to the same changes. However, at revisions 1.006 and 1.044 we observe changes into different directions, too.

Finally, coupling refers to the flow between different schemas in the specification. Though the value fluctuates, the developers managed to keep coupling quite stable on the long run. Fig. 4 also shows that the value fluctuates with the values of cohesion, but not necessarily to the same extent, and not necessarily inversely (as would be assumed to be normal).

Though with the *WSDL* specification there is only one experimental subject, the results seem to substantiate that the measures are suitable for assessing this Z specification. The measures called deterioration and relative deterioration reflect the aging of the system quite well, and the measures for coupling and cohesion (a) do indicate structural changes and (b) also seem to explain some of the semantic changes in the specification.

5 An Extended Model of Evolution

As has been shown in Section 4.3, specifications keep on changing. Either one is still in the process of finding the most suitable version for our requirements, or one is modifying them due to changes in our projects' software systems. With that, a second look at the software evolution model in Fig. 1 is quite helpful – as one comes down to the following statement so far:

There is also evolution before the birth of the running version of the software system.

Fig. 5 tries to exemplify this for the initial and evolutionary versions of the software. In this figure the original model has been extended by refining the boxes of the evolutionary versions. Documents and requirements have been added so that formal specifications are made explicit (as they do belong to the set of necessary documents). They are, depending on the changing requirements, also changed. These changes either happen before one has a first running version of the software or afterwards.

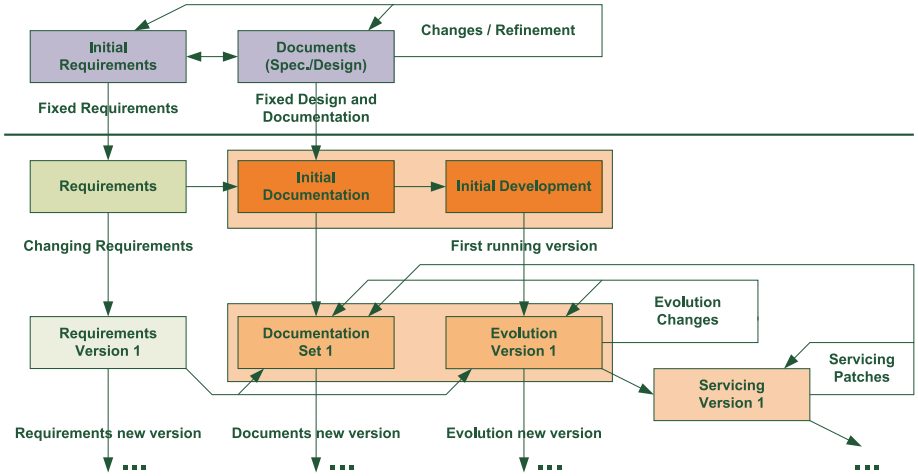

Fig. 5. A refined and extended look at the versioned stage model. Starting with a first set of initial requirements several versions of documents are created. Requirements are refined, and formal specifications are (among other design documents) also changed and modified. When the design is fixed, development is about to begin. Due to evolutionary changes after this phase, the existing documentation – including specifications and design documents – is (and has to be) changed, too. The term "evolutionary change of a formal specification" is used in a rather general sense. Apart from the classification of system types of Lehman [14], the figure illustrates that essential changes might happen to documents before and after delivery.

The implications of this (refined) picture are manifold and should be considered when using formal specification languages in the software development lifecycle:

- Suitable size- and quality-based measures should be defined. This ensures that changes in the various documents – including formal specifications – can be detected and assessed. The slice-based measures introduced above are just an example of how it could be done for a specific specification language. For other languages the ideas might be reused. It might also be necessary to define new measures. However, the crucial point is that there is a measurement system around.
- Points of measurement should be introduced at every change/refinement loop. This ensures that the effects of changes can be assessed, and that the further direction of the development can be steered. The example of *WSDL* shows that already during the initial development changes have effects and that it takes effort to keep a specification constantly at a pre-defined level of quality. One can assume that *WSDL* is not an exception and that the observation also holds for other specification documents. By making use of a measurement system one is at least on the safe side.
- The terms "Fixed Design and Documentation" just designate the conceptual border between the initial development phase and the first running version. Nevertheless, changes to the documents happen before and after this milestone in the project (as evolution is about to begin). The previously introduced measure points should also be defined for the evolutionary phases, and measures should be collected during all

the evolutionary changes and servicing activities (influencing the documents and specifications).

Basically, the extended model of software evolution makes one property of specification (and other documents) explicit: they are no exception to aging. With this, it is obvious that measures, at the right point and extent, help in answering the question of what happened and, eventually, of what can be done for preventing unwanted deterioration effects.

6 Conclusion

We started with the observation that formal specifications – documents important in the very early phases of software development and later on during maintenance – might be changed more often than commonly expected and that the changes are not necessarily positive. We wanted to verify this observation, and so we mapped existing semantic-based measures to Z specification and used them to analyze a large real-world specification. The lessons learned so far are manifold.

1. Firstly, there was no suitable measurement system around. In order to understand and to assess changes, new measures had to be developed. These measures, carefully mapped to Z specifications, have been evaluated by making use of a large set of sample specifications. They are maybe not representative for all specifications around, but the statistical tests helped us in gaining at least basic confidence in the results for Z.
2. Secondly, changes onto formal specifications definitely might influence the values of the measures and there is the chance that their effects are underestimated. There is no model that enunciates this situation, which is also the reason why we borrowed from the model of evolution and refined it to cover the phases before, within, and after initial development. A closer look at the evolution of the *WSDL* specification seems to confirm the observation mentioned at the beginning of the paper: formal specifications are not just written once. They are modified, are extended, and they age.
3. Finally, the measures of coupling and cohesion (and with them deterioration and relative deterioration) seem to be a good estimate for a qualitative assessment of a specification. They are easy to calculate and seem to point out a possible loss in quality.

With that, we are able to answer our two questions that have been raised at the end of Section 1: specifications evolve and this evolution can be observed by simple semantics-based measures. The refined and extended model of evolution as presented in Section 5 is a good image of what happens when developing our systems.

The results of this contribution are interesting insofar as it turned out that, for the full benefits of a formal software development process, it makes sense to permanently take care of the quality of the underlying formal specification(s). Even when declarative specification languages are used, this can easily be done by defining suitable measures

and by using them to constantly monitor the quality of the whole system. The goals for future work now include (a) taking a closer look at other formal specifications in order to verify and consolidate the findings, (b) investigating the correlation of the specification measures to code-based measures in order to come up with different prediction models, and (c) incorporating the refined model of software evolution into a formal software development process model that also considers cultural differences between the different stakeholders in a project.

Overall, the results are encouraging. Formal specifications are not necessarily restricted to the early phases of a software development process. When treated carefully (and kept up to date) they may help us in producing software systems that can be trusted, even when changed.

Acknowledgment

I am grateful to the reviewers of the FM 2011 conference and to my colleagues at AAU Klagenfurt, especially to Prof. Mittermeir, who helped me with fruitful discussions and reflections on this topic.

References

1. Bennet, K., Rajlich, V.: Software Maintenance and Evolution: a Roadmap. In: ICSE '00: Proceedings of the Conference on The Future of Software Engineering. pp. 73–89. ACM New York, NY, USA (2000)
2. Black, S., Boca, P.P., Bowen, J.P., Gorman, J., Hinchey, M.: Formal Versus Agile: Survival of the Fittest. IEEE Computer 42(9), 37–54 (September 2009)
3. Bollin, A.: Specification Comprehension – Reducing the Complexity of Specifications. Ph.D. thesis, AAU Klagenfurt (April 2004)
4. Bollin, A.: Concept Location in Formal Specifications. Journal of Software Maintenance and Evolution – Research and Practice 20(2), 77–105 (2008)
5. Bollin, A.: Slice-based Formal Specifiation Measures – Mapping Coupling and Cohesion Measures to Formal Z. In: Muñoz, C. (ed.) Proceedings of the Second NASA Formal Methods Symposium. pp. 24–34. NASA/CP-2010-216215, NASA, Langley Research Center (April 2010)
6. Carrington, D., Duke, D., Hayes, I., Welsh, J.: Deriving modular designs from formal specifications. In: ACM SIGSOFT Software Engineering Notes. vol. 18, pp. 89–98. ACM (December 1993)
7. Chang, J., Richardson, D.J.: Static and Dynamic Specification Slicing. Tech. rep., Department of Information and Computer Science, University of California (1994)
8. Clarke, E.M., M.Wing, J.: Formal Methods: State of the Art and Future Directions. Tech. rep., Carnegie Mellon University, CMU-CS-96-178 (1996)
9. Collins, B.P., Nicholls, J.E., Sorensen, I.H.: Introducing formal methods: the cisc experience with z. In: Mathematical Structures for Software Engineering. pp. 153–164. Clarendon Press Oxford (1991)
10. Harman, M., Okulawon, M., Sivagurunathan, B., Danicic, S.: Slice-based measurement of coupling. In: Proceedings of the IEEE/ACM ICSE workshop on Process Modelling and Empirical Studies of Software Evolution. Boston, Massachusetts. pp. 28–32. IEEE Computer Society, Los Alamitos, CA, USA (1997)

11. Hierons, R.M., Bogdanov, K., Bowen, J.P., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., Lüttgen, G., Simons, A.J.H., Vilkomir, S., Woodward, M.R., Zedan, H.: Using formal specifications to support testing. ACM Comput. Surv. 41(2), 1–76 (2009)
12. Hinchey, M., Jackson, M., Cousot, P., Cook, B., Bowen, J.P., Margaria, T.: Software engineering and formal methods. Communications of the ACM 51(9), 54–59 (September 2008)
13. Jackson, D.: Software Abstractions - Logic, Language, and Analysis. The MIT Press, Cambridge, Massachusetts (1996)
14. Lehman, M.M.: On understanding laws, evolution, and conservation in the large-program life cycle. Journal of Systems and Software 1(1), 213–221 (1979)
15. Meyers, T.M., Binkley, D.: An Empirical Study of Slice-Based Cohesion and Coupling Metrics. ACM Transactions on Software Engineering and Methodology 17(1), 2:1–2:27 (December 2007)
16. Mittermeir, R.T., Bollin, A.: Demand-Driven Specification Partitioning. Lecture Notes in Computer Science 2789(2003), 241–253 (2003)
17. Mittermeir, R.T., Bollin, A., Pozewaunig, H., Rauner-Reithmayer, D.: Goal-Driven Combination of Software Comprehension Approaches for Component Based Development. In: Proceedings of the ACM Symposium on Software Reusability - SSR01. Software Engineering Notes, vol. 26, pp. 95–102. ACM Press (2001)
18. Nogueira, J.C., Luqi, Berzins, V., Nada, N.: A formal risk assessment model for software evolution. In: Proceedings of the 2nd International Workshop on Economics-Driven Software Engineering Research (EDSER-2) (2000)
19. Oda, T., Araki, K.: Specification slicing in a formal methods software development. In: 17^{th} Annual International Computer Software and Applications Conference. pp. 313–319. IEEE Computer Socienty Press (November 1993)
20. Ott, L.M., Thus, J.J.: The Relationship between Slices and Module Cohesion. In: 11th International Conference on Software Engineering. pp. 198–204. IEEE Computer Society, Los Alamitos, CA, USA (1989)
21. Pirker, H., Mittermeir, R., Rauner-Reithmayer, D.: Service Channels - Purpose and Tradeoffs. In: COMPSAC '98 Proceedings of the 22nd International Computer Software and Applications Conference. pp. 204–211 (1998)
22. Roberto Chinnici, S.M., Moreau, J.J., Ryman, A., Weerawarana, S.: Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. http://www.w3.org/TR/wsdl20 (2007)
23. Ross, P.E.: The Exterminators. IEEE Spectrum 42(9), 36–41 (September 2005)
24. Samson, W., Nevill, D., Dugard, P.: Predictive software metrics based on a formal specification. In: Information and Software Technology. 5, vol. 29, pp. 242–248 (June 1987)
25. Vinter, R., Loomes, M., Kornbrot, D.: Applying software metrics to formal specifications: A cognitive approach. In: 5th International Symposium on Software Metrics. pp. 216–223. IEEE Computer Society, Bethesda, Maryland (1998)
26. Weiser, M.: Program slices: formal, psychological, and practical investigations of an automatic program abstraction method. Ph.D. thesis, University of Michigan (1979)
27. Weiser, M.: Program slicing. In: Proceedings of the 5^{th} International Conference on Software Engineering. pp. 439–449. IEEE Press, Piscataway, NJ, USA (1982)
28. Woodcock, J., Davis, J.: Using Z: Specification, Refinement, and Proof. Prentice-Hall International Series in Computer Science, Prentice Hall, Hemel Hempstead, Hertfordshire, UK (July 1996)
29. Wu, F., Yi, T.: Slicing Z Specifications. ACM SIGPLAN Notices 39(8), 39–48 (2004)