The Three Cs of Requirements: Consistency, Completeness, and Correctness

Didar Zowghi[1], Vincenzo Gervasi[2]

[1] Faculty of Information Technology
University of Technology, Sydney
didar@it.uts.edu.au

[2] Dipartimento di Informatica
University of Pisa
gervasi@di.unipi.it

Abstract. The initial expression of requirements for a computer-based system is often informal and possibly vague. Requirements engineers need to examine this often incomplete and inconsistent brief expression of needs. Based on the available knowledge and expertise, assumptions are made and conclusions are deduced to transform this "rough sketch" into more Complete, Consistent, and hence Correct (the three Cs) requirements. This paper addresses the question of how to characterize these properties in an evolutionary framework, and what relationships link these properties to a customer's view of correctness. Moreover, we describe in rigorous terms the different kinds of validation checks that must be performed on different parts of a requirements specification in order to ensure that errors (i.e., cases of inconsistency and incompleteness) are detected and marked as such, leading to better quality requirements.

1 Introduction

Software development is typically commenced when a problem is identified that may require a computer-based solution. The expression of the requirements for the new system is often informal and possibly vague, as Jackson puts it [10], a "rough sketch". Requirements engineers need to examine this incomplete and often inconsistent brief expression and based on the available knowledge and expertise, and possibly on further investigation, to transform this "rough sketch" into a correct requirements specification. The requirements are then presented to the problem-owners for validation. As a result, new requirements are identified that should be added to the specification, or some of the previously stated requirements may need to be deleted in order to improve it. So, at each step of the evolution of requirements, the specification can loose requirements as well as gain some. One of the critical tasks of requirements engineers in this process is to ensure that requirements specification at each step remains correct, or at least that errors are found as early as possible, their sources identified, and their existence tracked for future discussion.

It is frequently the case that in an attempt to maintain consistency within the requirements we remove one or more requirements from the specification and fail to preserve its completeness. Conversely, when we add new requirements to the specification to make it more complete, it is possible to introduce inconsistency in the specification. In this paper we argue that there is an important causal relationship between **C**onsistency, **C**ompleteness and **C**orrectness (the three **Cs**) of requirements. Increasing the completeness of a requirements specification can decrease its consistency and hence affect the correctness of the final product. Conversely, improving the consistency of the requirements can reduce the completeness, thereby again diminishing correctness. We ask the question: "Is it impossible to adequately address all three Cs simultaneously?" This phenomenon needs to be investigated further and guidelines have to be developed for requirements engineers as to which of these properties is of higher priority to maintain at each step of requirements evolution.

Correctness by itself is a vague concept. We can consider correctness from at least two different perspectives:

(1) From a formal point of view, correctness is usually meant to be the combination of consistency and completeness. Consistency refers to situations where a specification contains no internal contradictions, whereas completeness refers to situations where a specification entails everything that is known to be "true" in a certain context. We will be more specific when we refer to correctness in the next section, but for the moment let us emphasize that consistency is an internal property of a certain body of knowledge, whereas completeness is defined with respect to an external body of knowledge.

(2) From a practical point of view, however, correctness is often more pragmatically defined as satisfaction of certain business goals. This indeed is the kind of correctness which is more relevant to the customer, whose goal in having a new system developed is to meet his overall business needs.

In this paper we investigate what is the relationship between these two notions of correctness, and what kind of arguments can be made in support of the correctness of a specification. This question will lead us to explore how consistency and completeness affect these two notions.

Davis states that *completeness* is the most difficult of the specification attributes to define and *incompleteness* of specification is the most difficult violation to detect [4]. According to Boehm [2], to be considered complete, the requirements document must exhibit three fundamental characteristics: (1) No information is left unstated or "to be determined", (2) The information does not contain any undefined objects or entities, (3) No information is missing from this document. The first two properties imply a closure of the existing information and are typically referred to as *internal completeness*. The third property, however, concerns the *external completeness* of the document [3]. External completeness ensures that all of the information required for problem definition is found within the specification. This definition for external completeness clearly demonstrates why it is impossible to define and measure absolute completeness of specification because how could analysts know with certainty what is missing from the specification when they do not even know what it is that they are looking for in the first place. Clearly one of the available techniques that could assist

in the determination of external completeness of the specification is *domain modeling*.

2 An Evolutionary Model of Requirements Correctness

As mentioned above, completeness is a relative measure and may be determined only in relation to an external reference; it follows from (1) above that correctness in turn needs to be considered with respect to such an external reference.
According to Jackson [10,11], as illustrated in Figure 1, the role of requirements (R) in software engineering is to state relationships that are desired to hold between elements of a certain real world domain (D). Conversely, the role of a specification (S) is to provide instructions for a machine that has an interface to D so that the properties required in R hold.

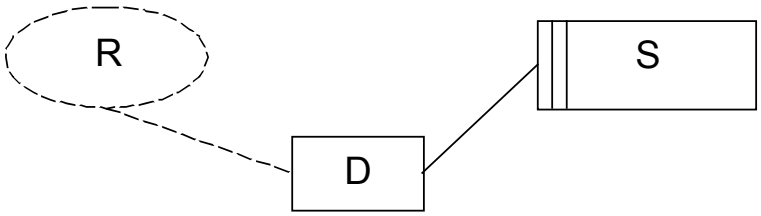

Figure 1 A simple diagram showing the relationship between Requirements, Domain, and Specification

Formally, we can write $S \cup D \models R$. In informal terms, this means that – given the assumption that the machine will perform as instructed by the specification, and that our model of the domain faithfully predicts how the real world will behave – what we know about the domain, together with what we know about the physical interfaces of the machine, will make R true in the end. This formula allows us to discuss the completeness of S with respect to R, but not the completeness (and thus the correctness) of R in isolation. Moreover, this relation must not be regarded as a method to synthesize S and D given R. Rather, it should be considered like a proof obligation that must be discharged if we want to prove the correctness of S. Indeed, $S \cup D \models R$ can be seen as proving that S and D together are complete with respect to R – that is, nothing that is required (by R) to hold is left out of either S or D. Also as part of the correctness proof, $S \cup D$ must be shown to be consistent (unless we do not have any requirement at all – in that case, even an inconsistent specification could be considered complete).

In informal terms, proving that $S \cup D$ is consistent can be thought of as ensuring that we are not asking the machine (in S) to perform something that is not possible in the domain (as stated in D). According to (1), proving both completeness and consistency will prove the correctness of S with respect to R and D: in essence, this is say-

ing that the specification we have developed satisfies our requirements in the given domain.

The problem of the correctness of R can only be formulated in a more complex setting, depicted in Figure 2, that provides an external reference for proving completeness. This setting presents an *evolutionary* view on the requirements. Several revisions of the requirements are considered, each one serving the role of a specification with respect to the previous one. This situation may be found in practice when we consider the common case of a product family undergoing several release cycles, but also, at a finer grain, inside a single release cycle. In fact, requirements are rarely – if ever – created all at once. Rather, they are usually obtained by progressively evolving a previous version of the same requirements, in order to reflect an increased understanding of the customer's needs. Furthermore, the domain gets evolved in a similar way, based on a deeper investigation about the relevant properties of the real world within which the system will operate.

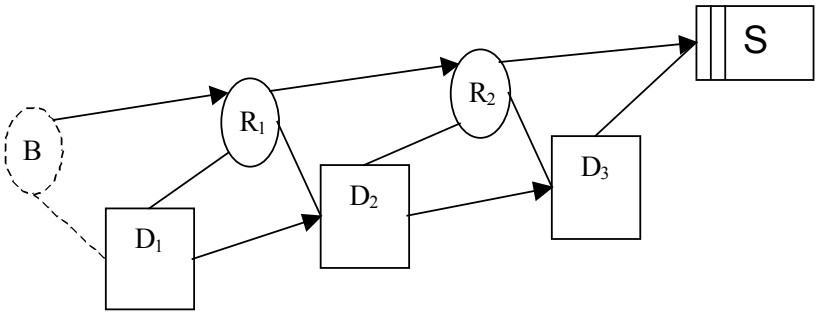

Figure 2 Relationship between S, D, and R in an evolutionary framework

As shown in Figure 2, we assume that at the beginning of this chain of evolution we have a statement of the business needs of the customer (B), and at the end of the chain we have a final specification that can be used for the implementation of the system. We will assume that the burden of proving the correctness of B with respect to the customer's *real* needs is on the customer, or on the analyst that performed the initial elicitation (or market analysis for the case of shrink-wrapped software). As an example, B might contain statements like "We need to reduce the time needed to deliver the product after a customer's order has been received". The correctness of such a statement, e.g., how effective such a measure would be to ensure bigger profits at the end of the quarter, is better left to management studies, and is in our opinion not a subject for requirements engineering. Thus, B provides a basis to avoid infinite regression, and serves as a reference point for the second definition of correctness that we gave in the previous section. Of course, the customer can change the business goals at any time, but then conceptually the evolution cycle will have to start again. In reality, however, in those circumstances we expect that the requirements engineer will be able to reuse most of the domain and requirements information previously captured.

In Figure 2, the arrows represent the evolutionary steps that lead to improved requirements and domain descriptions. In our view, requirements evolution needs not be *monotonic;* in fact, we expect that during the analysis process, and with a better understanding of the domain and of the customer's goals, the requirements analyst can change his mind about the desired behaviour of the system, adding or dropping requirements at any time. Thus, the arrows going from R_i to R_{i+1} do not represent *strict refinement*, but simply *change*.

On the contrary, we do assume that domain refinement is monotonic. Domain refinement occurs, among other cases, when new properties of the domain are incrementally uncovered as we come across requirements that may be in need of additional information from the domain to make them fully understood. Monotonicity in this context means that subsequent revisions of D can be more detailed or include more facts that previous versions, but they cannot contradict what was already known to be factually true of the real world. Naturally, this applies to observations made, and not to rules inducted by generalizing the observations For example, we could observe that when we send some text to a printing device, the text gets printed, and infer that this is true of any text – a knowledge that would enter our initial domain model. Later, we could discover instead (by digging into the manuals of by means of further experimentation) that certain sequences are interpreted as commands (e.g., clearing the current print buffer) and not printed. We could then refine our domain model by adding appropriate exceptions to the general rule. This refinement would make the model more faithful, but would not contradict the original observation that the text sent in our first experiments was printed out.

Of course, this monotonicity assumption does not include the situation where contradictions between two successive Ds are due to *inaccurate descriptions* (e.g., clerical errors) of the real world. As we will see, this fact does not invalidate our argument; and we simply assume that such inaccuracies are corrected in the course of the development. However, if we decide to treat requirements model R_i as a logical theory, we can then follow the distinctions made in the *theory change* literature [12] between "revising" and "updating" a body of knowledge. The former is used when we are obtaining new information about a *static* world where newer results may contradict the old ones while the latter consists of bringing the information up to date when the world described by it changes (i.e. a *dynamic* domain).

In formal terms, the intertwining between requirements evolution and correctness can be expressed as follows (for simplicity, we will use the name R_0 for B, and R_{n+1} for S). We have assumed that R_0 is correct – that is, complete with respect to the real user business needs and internally consistent.

At each step, we have to prove that

$R_i \cup D_i \models R_{i-1}$ (completeness of R_i and D_i w.r.t. R_{i-1}) and
$R_i \cup D_i \not\models \perp$ (consistency of $R_i \cup D_i$).

Moreover, at each step $D_i \models D_{i-1}$ (monotonicity for domain descriptions). It immediately follows that:
$R_i \cup D_i \models R_{i-1} \cup D_{i-1}$.

This can be interpreted as the fact that, even if the various versions of the requirements are not in a refinement relationship, the union of the requirements and of the domain descriptions are. It follow by simple induction that

$R_{n+1} \cup D_{n+1} \vDash R_0 \cup \{\ \}$

or, in other terms, $S \cup D_{n+1} \vDash B$

That is, the final specification, deployed in the domain described by our final domain model, satisfies the customer's business goals. This last expression is indeed our definition (2) of correctness.

3 Example and Discussion

Let us illustrate the concepts presented above with the following example.

Consider the high-level business goals of a shop owner. As part of these goals, she wants an automatic sliding door to be installed at the entrance, to encourage prospective customer to enter the shop, and improve the image of her business. We could state that goal as $B=R_0=$\{when a customer comes near the entrance, the door shall open\}. At the beginning of the development process, we have an empty domain model (i.e., the description of the domain is an empty set), and of course B cannot be realistically implemented as it is on a physical machine. As part of the evolutionary approach presented above, we have to progressively move parts of our proof obligations from R to D. As a first domain description we could have $D_1=$\{when a person comes near the entrance, a presence sensor gets activated\} and $R_1=$\{when the sensor gets activated, the door shall open\}. Notice that R_1 and B are not in any formal entailment relationship (as expected, due to the non-monotonicity of refinement on R).

However, if we try to prove that $R_1 \cup D_1 \vDash B$, we fail. The reason is that B asks for *customers* to cause the opening of the door, whereas our investigation in the domain of presence detectors has shown that sensors can only detect *persons*[1], and are not able to discriminate between (past or future) customers and people just passing by.

In this case, the failure at proving consistency (and thus correctness) is due to an infeasible statement of business needs. We could (greatly) enhance our domain, investigating biometrics, face recognition, and other advanced surveillance techniques in order to recognize potential customers even before they enter the shop, but let us assume instead that a weakened version of B (concerning people, not customers) is acceptable in this case. Then, $R_1 \cup D_1$ is consistent and complete with respect to the new B. We can repeat the process, obtaining

[1] In fact, such sensors can generally only sense *movement* in a certain area, not even *persons*. This is a nice and desirable feature when we consider parents pushing their strollers towards the entrance, but an inconvenience when we think of stray dogs and cats. For simplicity, we ignore the issue in this example: selecting the proper kind of sensors lies in the domain of system engineering, whose interplay with requirements engineering has been investigated in other works [14]

$D_2 = D_1 \cup$ {when a sliding door's motor is turned on, the door opens} and

R_2={when the sensor gets activated, the door's motor shall be turned on}.

Again, we can prove that $R_2 \cup D_2$ is consistent, and complete with respect to R_1. Moreover, we can prove that $D_2 \models D_1$ (actually, it contains it). Also, $R_2 \not\models R_1$; once again we are confronted with non-monotonic evolution of the requirements. Iterating the process, we will eventually come to a specification like S={when a signal is detected on the input line associated with the door's presence sensor, establish +5V on the output line associated with the door's motor}, with a correspondingly detailed domain description.

If we have proved consistency and completeness at each step, we obtain by induction that this specification, together with its domain description, are correct with respect to B (according to (1)). Yet, this is exactly what correctness in definition (2) is all about. Definitions (1) and (2) actually express the same basic notion. The intuitive idea of correctness from the customer's point of view given in (2) is just a coarse-grain view of the more formal and precise definition given in (1).

It is important to stress that the framework presented here identifies which kind of consistency and completeness checks must be performed to verify correctness, but do not prescribe how to handle them, and in particular do not impose that the requirements and domain model themselves must stay correct at all times. It is a well-known fact of life that requirements are often incomplete and inconsistent during most of their life. The three proof obligations discussed above (completeness of R_i and D_i w.r.t. R_{i-1}, consistency of $R_i \cup D_i$, and monotonicity of domain descriptions) can be interpreted as validation checks that can (and should) be made during requirements evolution in order to identify and expose possibly latent errors. Once such errors are exposed, they can be tolerated (as advocated by [1,5,6,9,13]), if they reflect a genuine conflict of goals or simply there is not enough information available yet to decide how to correct them, or corrected immediately by changing the relevant requirements or domain model elements. In any case, an informed decision can be taken only after such cases have been identified and carefully analyzed.

We can now look back at the question we posed in the introduction. As the brief discussion above and the illustrative example have shown, we believe that the two notions of correctness are indeed closely related to each other. It is thus important to explore the more formal treatment implied by definition (1), because that will provide us with the pragmatically more relevant correctness in (2). We can state that a formal treatment of consistency, completeness and correctness in requirements specifications will allow us to satisfy the business goals of the customer (beside being a valuable contribution by itself).

4 The Three Cs in Practice

Although it is an advantage to have a formal proof of correctness of a specification (e.g. as in [8]), it may not be practical or may be too costly to do so. Indeed, in many cases such proofs can be carried out by informal (but rigorous) inspections of the

requirements and domain descriptions. The decision as which is the more appropriate course of action depends on the degree of risk the stakeholders are prepared to take. In safety critical software, for example, formal descriptions and proofs are usually deemed necessary, while in business applications other factors like time to market or development cost can be more important. Moreover, it is often the case that the requirements are vague at the beginning, and gain formality while their evolution proceeds. It is thus not uncommon for the respective proofs to be rather informal at the beginning, while becoming more and more formal as the requirements and domain descriptions themselves become more formalized.

Of course, the main advantage of the approach we have proposed lies in the capability of immediately identifying those changes in the requirements or in the domain model that might introduce errors in the specification, thus achieving more precise verification and validation of the requirements. This kind of checks are much more efficient if performed in a continuous way (i.e., each edit action on the requirements or on the domain model triggers a check of the three properties described in Section 2) and automatically, by using appropriate tools [7,16]. Automation can be achieved by directly writing the requirements in a formal language that allows automatic theorem proving (e.g., propositional logic or Datalog), or by using controlled natural language and providing a suitable translation layer (as done, for example, in our previous work cited above) instead. Naturally, for reasonably simple specifications (i.e., small, well written, and easily navigable), and given enough resources, manual verification is also possible and – as said above – can even be more convenient. However, it is difficult to guarantee constant reliability of these manual checks, so automation should be sought whenever possible.

The ability to evaluate the impact that a new proposed requirement may have on the correctness of a specification is also of great importance in other cases. For example, the requirements prioritization used during negotiations could incorporate an indication of how much and to what extent each of the requirements being discussed contributes to the overall completeness of the specification. This could be achieved by examining and ranking the dependencies among individual requirements and also by building requirements into clusters that contribute to reach a specific business goal.

Also, at each step of evolution, it is important to identify any emerging inconsistencies resulting from adding new requirements. By providing automated tools (e.g. [16]), that suggest alternative solutions on how to manage inconsistencies, together with the corresponding measures of completeness for each alternative, the requirements engineer could be guided on what course of action to follow to maintain a balance of completeness and consistency and hence correctness of requirements specifications.

5 Conclusion and Future Works

In this paper we provided a theoretical underpinning for the pragmatic view of correctness, thus introducing more rigor into the process of requirements evolution. In detail, we have described which kind of proofs must be carried out at each step during

the evolution of the requirements in order to ensure that the final specification of a software system satisfies the business goals of its customer. We have also proposed various ways in which our model can be applied to real-life circumstances, both for validation purposes and as a supporting technique during requirements negotiation and prioritization.

Furthermore, we hope that this work will bring to the attention of requirements engineers the importance of considering the three Cs (and their often competing nature) at each step of evolution, rather than as one-shot properties to be checked only as part of the final validation of the specification.

Our vision is a starting point for a new line of research that aims to provide practical tools and methods that alleviates the burden of providing proofs at each stage of system evolution. Common lore has practitioners voicing unflattering judgments like: "Computer scientist… Their pronouncements are more relevant to Zen than to the no-nonsense business of building useful … programs and systems. They have no answer to real life problems like users who change their minds or requirements that are in a constant state of flux." (Anonymous, cited in [10] page 113). We believe instead that the integration of rigorous and formal results in an evolutionary model of requirements development helps in reaching those very no-nonsense business goals that were called for in the statement above.

In related research we have developed automated tools supporting consistency checking in natural language requirements [16] as well as in formal logic [15]. It is our intention to extend our approach to also support completeness checking at each step of evolution, thus providing automated proofs of correctness as outlined in Section 2. Such automated support will allow us to test the validity of our argument by applying it in a case study over an entire release of a product family, that we plan to jointly develop with an existing industry partner.

References

1. Balzer R., "Tolerating inconsistency", Proc of the 13th IEEE International Conference on Software Engineering (ICSE 13), pages 158-165, Austin, Texas, (1991), IEEE Comp Soc Press.
2. Boehm BW., "Verifying and validating software requirements and design specifications", IEEE Software, 1(1):75-88, (1984).
3. Cordes DW., Carver DL., "Evaluation methods for user requirements documents", Information and system Technology, 31(4): 181-188, (1989).
4. Davis AM., "Software Requirements: Analysis and Specification", Prentice Hall, second edition, (1993).
5. Easterbrook S., Nuseibeh B., "Managing inconsistencies in an evolving specification", Proc of the 2nd International Symposium on Requirements Engineering (RE95), pages 48-55, York, England, (1995).
6. Finkelstein A., Gabbay D., Hunter A., Kramer J., and Nuseibeh B., Inconsistency Handling in Multi perspective specifications. IEEE Transactions on Software Engineering, 20(8):569-577, (1994).

7. Gervasi V., and Nuseibeh B., “Lightweight validation of natural language requirements”. Software: Practice & Experience, 32(2):113-133, February (2002).
8. Heimdahl M. P. E. and Leveson N. G., “Completeness and Consistency in Hierarchical State-Based Requirements”, IEEE transactions on Software Engineering, Vol 22, No. 6, June (1996).
9. Hunter A., Nuseibeh B., “Managing Inconsistent Specifications: Reasoning, analysis and action., ACM Transaction on Software Engineering and Methodology, October (1998).
10. Jackson M., “Software Requirements & Specifications: a lexicon of practice, principles and prejudices”, Addison Wesley, Great Britain, (1995).
11. Jackson M., “Problem Frames: Analyzing and structuring software development problems”, Addison Wesley, Great Britain, (2001).
12. Katsuno H., and Mendelzon, A.O., “On the difference between updating a knowledge base and revising it”, Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning, (1991).
13. Nuseibeh B., "To be and not to be: on managing inconsistency in software development", in IWSSD-8, pages 164-169, IEEE Comp Soc Press (1996).
14. Stevens R., “Systems Engineering – Coping with Complexity”, Prentice Hall, (1998).
15. Zowghi D., and Offen RJ., “A Logical Framework for Modelling and Reasoning about the Evolution of Requirements”, Proceedings of the Third IEEE International Symposium on Requirements Engineering (RE97), Pages 247-259,(1997).
16. Zowghi D., Gervasi V., and McRae A., “Using Default Reasoning to discover inconsistencies in Natural Language Requirements”, Proceedings of the 8th Asia-Pacific Software Engineering Conference, December (2001), Macau, China.