On the Interplay Between Consistency, Completeness, and Correctness in Requirements Evolution

Didar Zowghi

Faculty of Information Technology
University of Technology – Sydney
Australia

Vincenzo Gervasi

Dipartimento di Informatica
University of Pisa
Italy

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 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.

Key words: Software requirements, consistency, completeness, evolutionary correctness assurance.

Email addresses: `didar@it.uts.edu.au` (Didar Zowghi), `gervasi@di.unipi.it` (Vincenzo Gervasi).

Preprint submitted to Elsevier Science *1 April 2003*

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 [1], 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 alternatively that errors (i.e., violations of correctness properties) 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 Consistency, Completeness and Correctness (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: “What are exactly the interplays among the three Cs?” These relationships need to be investigated further and guidelines have to be developed for requirements engineers as to which is the best way to check and maintain these properties at each step of requirements evolution.

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

(C1) 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 desired to hold in a certain context. We will be more specific when we refer to correctness in Section 3, 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.

(C2) From a practical point of view, however, correctness can be 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. These questions will lead us to explore how consistency and completeness affect these two notions.

Our work does not provide a universal solution to the difficult problem of proving the correctness of requirements. Too many factors are involved in writing good requirements, and we believe any “silver bullet” for this problem to be illusory. Rather, the paper offers a better theoretical understanding of how consistency and completeness interact with correctness during requirements evolution, and presents a framework that can be applied, with a variety of formal and informal proof methods, to ensure that these properties are maintained (or recovered, if lost) during the evolution of the requirements. In our opinion, having a clear understanding of *what* to prove can be as important as finding the right correctness proof for each concrete case — both in practice and in education.

The paper is organized as follows. Section 2 presents an overview of the scientific literature concerning consistency, completeness, and correctness in requirements. Section 3 presents our framework for requirements correctness in an evolutionary setting, together with the formal definitions of the proof obligations that must be discharged in order to prove correctness at each step. It is followed by a complete example, showing how our framework is applied in a realistic (although simplified) case. Section 5 discusses the impact of our approach on current practice, and how other real-life problems relate to our model. Some conclusions, and ideas for future work, conclude the paper.

2 Related works

In this section we provide a short overview of the literature related to the concepts discussed in this paper.

2.1 Consistency

Consistency requires that no two or more requirements in a specification contradict each other [1] . It is also often regarded as the case where words and terms have the same meaning throughout the requirements specifications (consistent use of terminology). These two views of consistency imply that mutually exclusive statements and clashes in terminology should be avoided.

A wide variety of possible causes of inconsistency in software development have been identified in the research literature. For example Nuseibeh [2] views inconsistency as it arises between the views of multiple stakeholders in software development. Among the reasons for the occurrence of inconsistencies, he includes different language usage, varying development strategies, different views held by participants, and the degree of overlap that exists in the areas of concerns of different stakeholders. Nuseibeh defines an occurrence of an inconsistency very simply as the breaking of a rule, specifically a consistency rule that explicitly describes some form of relationship that must hold between two partial specifications. Easterbrook et al. [3] regard an inconsistency similarly as any situation in which two parts of a specification do not obey some relationship that should hold between them. Being very general, these definitions of inconsistency are not very informative: in fact, depending on the particular rules or relationships that are being considered, any defect in a specification (e.g., redundancy) could be termed as "inconsistency".

An important distinction is made between *inconsistency* and *conflict* in [4]. A conflict is the interference in the goals of one party that is caused by the action of another party. This view is shared by Robinson and Volkov [5] who state that a "conflict" is a *negative interaction* where an *interaction* is a dependency between two or more requirements and a conflict indicates that a requirement interferes with the achievement of another requirement. An inconsistency, on the other hand, occurs if a certain rule defined by the method designer is broken. A conflict occurs typically when one party makes changes that interferes with the other party's development. This means that conflicts do not necessarily result in any consistency rules being broken [4]. However, if a conflict is considered as a mere interference, in effect what is being expressed is that when a conflict arises, a fundamental rule of *non-interference* has been broken. In that case conflicts and inconsistencies are indeed equivalent, the only difference is in the type of rule that has been broken. Hence, any mechanism for detecting, resolving and managing inconsistency should effectively be able to apply to conflict management too.

The approach taken by Easterbrook and Nuseibeh expressed in the above defi-

[1] Or, more precisely, that it is never the case that the requirements cannot be all satisfied at the same time.

nitions, moreover, makes a major, incorrect, assumption that an inconsistency is necessarily a binary relation, that is, an inconsistency is caused by two parts of a specification. This problem has also been recently identified by van Lamsweerde et al. [6] who state that most current techniques for inconsistency handling in the current literature consider binary conflicts only.

Cugola et al. [7] distinguish between *deviations* and *inconsistencies*. They address inconsistencies and deviations that arise in software development processes and discuss ways in which inconsistencies between *performance state* and *enactment state* can often be avoided by preventing further development activities till some preconditions are forced to hold. Their logic-based technique is used to capture and tolerate some deviations from a process description during execution. In their approach some deviations can be tolerated as long as the correctness of the system is not affected. In this case deviations are recorded and propagated. A form of logical reasoning is carried out which identifies some possible sources of inconsistency. If these identified sources do interfere with the correctness of the system, then the incorrect data have to be fixed or the process model has to be changed.

A more comprehensive view of what consistency means in software development is adopted by Hangensen et al. [8]. Their proposed framework uses the notion of a “description” for structuring and representing information. A description is defined as some kind of object, which represents a piece of information and which could have any number of “interpretations” depending on the observers’ decision on which universe of phenomena are relevant for the observation. An actual subset of phenomena from this universe of phenomena covered by the description is defined by means of an interpretation. The consistency condition for two descriptions is defined as relations between interpretations of descriptions. Their techniques for consistency handling are modeled with respect to descriptions, interpretations and relations.

Other approaches that have emerged in the recent research literature attempt to accommodate inconsistent data in a database by paying due attention to the environment. For example Balzer [9] addresses the handling of certain kinds of inconsistency. He discusses the notion of *relaxing constraints* and *tolerating* inconsistencies and gives a simple technique to do this. He proposes modifying constraints to explicitly identify their violations with “pollution markers” acting as guards that can be used to modify the application’s behavior so that the inconsistency can be tolerated by screening code that is sensitive to the inconsistency or adjusting its behavior. He admits, however, that there is an implicit informal goal of *minimizing* both the number and the duration of inconsistencies and that they have to be resolved eventually and the sooner the better. This admission is a clear indication that no matter how useful or important inconsistencies are in requirements negotiations and discovery, they have to be resolved before software development can continue into the next stage, namely high-level design. Tolerating inconsistency, therefore, can only be considered as a temporary measure. According to Balzer who was the initial proponent of the notion of tolerating inconsistency in software development, the duration and the number of inconsistencies have to be minimized. Balzer's approach therefore does not provide any strategies for the effective management of inconsistency which includes the minimization of inconsistency.

Balzer further observes that his automatic repair technique only works for particular situations and is not generally applicable because there are potentially many ways to resolve inconsistencies [9]. He suggests additional *user input* to be sought to complement the automated strategies. In fact this appeal to *user intervention* for resolution of inconsistencies seems to be prevalent in many frameworks of this nature for inconsistency handling (e.g. [10–17]). Furthermore, Balzer's approach mainly addresses the down-stream activities of software development, namely handling inconsistency in design and code specifications and in databases. Many of the techniques, strategies and methods that were introduced initially for the down-stream activities of software development have often been adopted and tried by RE researchers but mostly with limited success. This is because RE activities are fundamentally different from the other activities in software development and there are many so called "softer" factors involved in the construction of specifications that are unique to RE.

Consistency in requirements models thus implies a lack of contradiction within the presented information. Both a direct refutation of previously stated requirement and an indirect denial of this description can constitute contradictions within the requirements model. Direct refutation represents statements within the model that are incompatible with each other. The truth of the first statement of a requirement directly negates the truth of the second statement. Moreover, information within the model can be refuted in an indirect manner. A given set of facts could establish a potential situation that, given the proper set of circumstances, would contradict other facts within the model. Therefore, establishing consistency within a requirements model is primarily a semantic task.

Requirements models include both explicit assumptions about the problem, application domain, and the machine as well as implicit consequences of these assumptions. A distinction is made between *explicitly expressed* requirements, which means that one's requirements involve an explicit mental representation whose content is the content of the requirements, and *implicit* requirements, which are derivable from the explicit requirements. In practice whether a statement is an implicit consequence is a matter of degree. The accessibility of a consequence depends on the complexity of finding a derivation. The more complex the derivation, the more inaccessible its consequence. If a derivation is too complex, its consequence is as inaccessible as if it were not implied at all [18].

In the same way, inconsistency can occur between explicit requirements (in this case, it is more akin to a conflict), or involve implicit requirements (and in this case, the cause may be an unforeseen interaction between requirements).

2.2 Completeness

While consistency has received greater attention by researchers in recent years, completeness has also been addressed in a number of studies. Completeness is an important component of software correctness, be it in a safety critical system, embedded system or any other type of systems. One of the causes of computer-related accidents and system failures is claimed to be the incompleteness (among other flaws) in the software requirements, and not coding errors [19]. *Completeness* is considered to be the most difficult of the specification attributes to define and *incompleteness* of specification the most difficult violation to detect [20].

According to Boehm [21], 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 [11]. 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. The only truly complete specification of something would be the thing itself. A compromising position would be to determine whether a specification is *sufficiently* complete. Decision on what is sufficient completeness would have to be defined with respect to the type of system being implemented. For example, in safety-critical systems sufficient completeness may be defined with respect to system safety design constraints as well as requirements derived from hazard analysis [22]. Clearly one of the available techniques that could assist in the determination of external completeness of the specification is *domain modeling.* This will be discussed in Section 2.4.

The importance of specification completeness in process-control applications has received a great deal of attention in recent years. For example, Jaffe et al have developed a set of formal criteria to identify missing, incorrect, and ambiguous requirements for process-control systems [23]. This work has been continued by Leveson in the design of formal specification languages [22]. Leveson argues that using a formal specification language alone will not lead to finding incompleteness in the specifications and states that the level of incompleteness discoverable is very much dependent on the features of the formal specification language. Throughout the development of various formal specification languages, Leveson has found that propositional logic notation does not scale well to complex expressions in terms of readability and to overcome this concern, she has developed a tabular representation of disjunctive normal form called AND/OR tables.

The notion of completeness is essentially related and tied to the purpose and goal of a system. That is, the completeness of a specification is relative to satisfying a certain set of goals. Letier and van Lamsweerde [24] suggest that goals must be made explicit in the requirements engineering process because goals drive the elaboration of requirements to support them and that they provide a criterion for measuring requirements completeness. They define the three Cs of requirements with respect to goal operationalization. In goal oriented RE such as I* [25] and AGORA [26], goal refinement and elaborations are used in the form of AND/OR graph to validate requirements specification for completeness.

Completeness of specifications has also been tied to their performance. For example, Woodside et al. [27] have investigated software specification completeness for its performance potential. These range from the execution cost of operation and details of deployment up to missing subsystems and layers. Hence, the "performance completion" of a specification implies that it contains enough information to evaluate the performance of the system, to the desired accuracy.

Most recent studies of completeness of specification in the literature are related to the application of some formal method for analysis and validation or often related to algebraic specifications [28]. For example Sheldon et al [29] have combined Z, Statecharts, and Stochastic Activity Network to validate a Natural Language software requirements specifications for completeness, consistency, and fault-tolerance.

The completeness of requirements specifications is not only with respect to formal logic or mathematical theory but also includes completeness in regards to socio-technical context of the system to be built. That is, human related aspects of completeness are important as well as application domain related issues of completeness. Completeness of a specification thus implies that all customer's needs will be met when the system is constructed. One's intuition about completeness frequently originates from one's sense of causation [30].

2.3 Correctness

Correctness of a requirements specification describes the correspondence of that specification with the real needs of the intended users much the same way that correctness of a piece of software refers to the agreement of the software part with its specification. Therefore, describing the software part X as being correct is identical to stating that software part X meets all the requirements imposed by the specifications [31]. Similarly, stating that the specification of a software component is correct is equivalent to saying that the specification meets (as far as possible) all the business goals stated by the customer.

One of the recent applications of formal methods in RE is that of Heitmeyer et al. [14] where the specification is expressed in SCR (Software Cost Reduction) tabular formal notation and an automated consistency checker is used to check the specification for syntax and type correctness, coverage, determinism, non-circular definitions and other application independent properties. Heitmeyer et al. provide a list of seven correctness properties derived from their formal requirements model in SCR formal notation that is represented in Finite-State Automaton (FSA), which produces externally visible outputs in response to changes in monitored environmental quantities. The main purpose of these checks is to determine if the requirements specification is well formed and so they constitute a form of static analysis. Although the notation and the consistency checking techniques seem promising, the requirements process they have adopted is, in their own words, "an idealization of a real-world process". This is because they assume that it is possible to express all of the requirements in their formal notation first, then run the automated consistency checker over it to identify all inconsistencies in linear time and for the *users* to finally *intervene* and correct the detected inconsistencies. But in the real-world RE is an evolutionary and incremental process and hence the inconsistency analysis which is part of this process must also be performed in an evolutionary and incremental manner. This means that consistency checking is part of the construction of the requirements specifications and should be performed in parallel.

2.4 Requirements and domain modeling

Domain modeling improves the communication between the problem owners and the requirements engineers. In many cases modeling the domain of application and developing a conceptual model of the environment leads the analysts directly to the requirements, since the goal of the software is to support a required mode of operation in the environment. This is particularly true of those projects whose purpose is to automate an existing manual system. Another very important role of domain modeling is associated with changing requirements. Since changes are inevitable in systems development, and these changes often originate from a change in the environment, by modeling the environment the requirements engineer can investigate potential changes and provide valuable information for the designers so that they can achieve appropriate modularity in design

Domain modeling is also used within RE to uncover the conceptual models of the participants and their relationships during requirements elicitation. The term domain modeling has been used in a variety of ways in the research literature but its common usage has been to model an application domain within which a family of related systems is constructed. The best description of what domain modeling and requirements modeling are really about is given, in our opinion, by Jackson and Zave [32,33]. They observe that part of the real world becomes the *environment* of a development project, because currently the problem-owners are dissatisfied with its behavior. Then a problem solver (developer) suggests the building of a computer-based *machine* such that when is introduced into the existing environment it would change the behavior of the environment to the satisfaction of the problem owners. Jackson and Zave state that the input and output of the machine are really part of the environment and from this perspective, they claim that *all* statements made during RE are about the environment. They do, however, provide a distinction by suggesting the use of two distinct grammatical moods in systems development [34]. The first mood is the *indicative* — that which is. The second mood is the *optative* — that which we would like to be. The indicative mood of systems development is also known as *domain analysis*.

Jackson refers to software development as building a machine simply by describing it [1]. He further states that *description* is at the heart of software development in the sense that programs are descriptions of machines; requirements are descriptions of the application domain and the problems to be solved there; and specifications are descriptions of the interface between the machine and the application domain. Some of the environment phenomena D are private to the domain, and internal machine phenomena M are private to the machine. Both D and M, however, have also shared phenomena, that can be sensed or caused by sensors and actuators. The machine's external behavior and properties are described in a specification S, expressed as a set of relationships over the shared phenomena. Requirements R are expressed as a set of relationships over the domain phenomena, that is, $D \cup S$. We will come back to these concepts in the next section.

An interesting application of formal methods to domain modeling and requirements engineering is that of Dines Bjorner's work on formalizing the domain model for railway systems using the RAISE (Rigorous Approach to Industrial

Software Engineering) Specification Language (RSL) [35]. Unlike many other formal methods, RAISE contains a method, a toolset as well as a specification language. It embodies logic and features of VDM, OBJ, CCS, CSP, and Standard ML. In a number of case studies, Bjorner shows how domain modeling is an extremely important part of requirements engineering, and how well-constructed domain descriptions can be even richer, more extensive, and more informative than the requirements themselves.

3 An evolutionary model of requirements correctness

We consider now the general problem of determining whether a given requirements specification is *correct* or not. As mentioned in Section 1, completeness is a relative property and may be determined only in relation to an external reference. It follows from our definition (C1) of correctness that correctness, in turn, needs to be considered with respect to such an external reference. However, if we look at the requirements analysis process as a whole, no such reference can be found — or, even if found, it is so removed in abstraction level from the final requirements specification that it is hardly of any use. Thus, the question arises: how can we discuss the correctness of requirements (and especially their completeness) in rigorous terms if we do not have a direct term of comparison?

In this section we present a framework for requirements evolution that allows us to show how a suitable external reference can be found at each step, and how this fact can be used to simplify and modularize correctness proofs, up to showing correctness of the final specification.

3.1 Requirements, domain, and specification[2]

According to Jackson [1,36], 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. Thus, both R and S predicates on entities in D, although with different intentions: R to state what is desired, S to instruct a machine to behave in such a way that R holds. In Figure 1, this is represented by the lines connecting R and S to D: the dashed

[2] In the following, we use standard notation from set theory and logic, applied to sets of statements. See Appendix A for a brief explanation of the notation used.

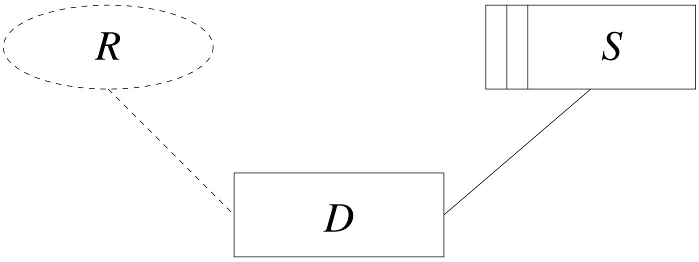

Fig. 1. A simple diagram showing the relationship between Requirements, Domain, and Specification

line means "optative predication", while the continuous line means "indicative predication".

Formally, we want to ensure that $S \cup D \models R$. 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 describes how the real world behaves — what we know about the domain, together with what we know about the physical interfaces of the machine, will satisfy R 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 [3].

In informal terms, proving that $S \cup D$ is consistent can be thought of as ensuring that we are not asking the machine (through S) to perform something that is not possible in the domain (as stated in D). According to (C1), proving both completeness and consistency will prove the correctness of S with respect to R and D: in essence, this is saying that the specification we have developed satisfies our requirements in the given domain, and hence S is correct. This, however, says nothing about the correctness of R.

3.2 On the evolution of R and D

The problem of the correctness of R (as opposed to that of the correctness of S discussed above) can only be formulated in a more complex setting, depicted in Figure reffig:evol, that provides an external reference for proving the completeness of R. This setting presents an evolutionary view on the require-

[3] Unless, of course, we do not have any requirement at all — in that case, even an inconsistent specification could be considered complete.

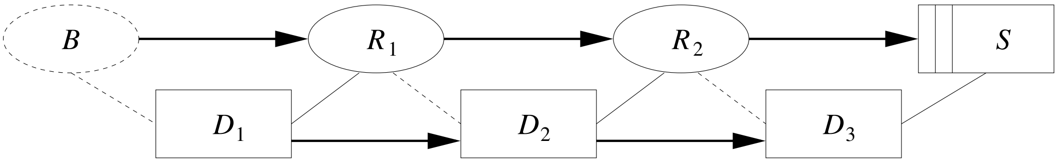

Fig. 2. Relationship between S, D and R in an evolutionary framework. Arrows represent evolution steps between successive versions of requirements and domain descriptions.

ments. 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.

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 ship a product after a customer's order has been received". The business-oriented view of correctness of such a statement, i.e., 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[4]. On the other hand, finding the best way to turn these business needs into a software system that satisfies them *is* a matter pertaining to requirements engineering, as is proving the correctness of the specification for such a system.

In our framework, B provides a basis to avoid infinite regression, and serves as a reference point for the second definition of correctness that we gave in Section 1. Of course, the customer can change the business goals at any time, but then conceptually the evolution cycle will have to start again. In practice, however, in those circumstances we expect that the requirements engineer will be able to reuse most of the relevant domain and requirements information

[4] In fact, this initial statement has nothing to do with software, and could be potentially satisfied by other means, e.g., by hiring more people or by outsourcing shipping operations completely.

previously captured, as we will show later.

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 behavior 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* from one version of the requirements to the next. In typical development processes, requirements evolution starts with a phase in which monotonic change is predominant, i.e., requirements are accumulated until the core set of needed features has been reached. After that, evolution becomes mainly non-monotonic: errors found in the requirements are corrected, marginal features are changed or discarded, experimental requirements are added and then retracted, etc.

On the contrary, domain refinement is often monotonic throughout the process. 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 do not contradict what was already known to be factually true of the real world. In other words, the domain *discovery* process yields essentially monotonic evolution. Non-monotonic refinement can also happen, for example when same aspect of the domain changes — in this case, the domain itself is evolving, whereas in the previous case it was our knowledge of the domain that was evolving.

This distinction between monotonic and non-monotonic refinement of requirements and domain descriptions is known in the *theory change* literature [37] as the distinction 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*).

3.3 Proving correctness

In this section we analyze which properties should be proved at each evolution step in order to guarantee correctness, both in sense (C1) and in sense (C2) from Section 1. We first consider how completeness can be proved in the presence of static domains or dynamic domains, assuming — without loss of generality — that requirements evolution is always non-monotonic. Then we show how discharging the proof obligations we have found for consistency and completeness, correctness in sense (C2) can be obtained.

To simplify the notation, we denote with R_0 the initial set of business needs B, with $D_0 = \emptyset$ the initial, empty domain description, and with R_{n+1} the final specification S. We assume that R_0 is "business-wise correct", that is, complete with respect to the real user business needs and internally consistent, as discussed above.

Notice also that, although we use the standard notation from logic, there is no constraint about the language used to express R and D, nor on the way entailment is proved. Indeed, requirements and domain description statements could be expressed in plain English instead of using logic formulas. In that case $\models$, too, must be interpreted as "any reasonable inference" among sets of English statements. We will return to the subject in Section 5.

We first introduce two lemmas that will allow us to simplify the most common cases for our main theorem:

Lemma 1 (Monotonic domain refinement) *Let us assume that we are performing an evolution step, from R_i and D_i to the subsequent versions R_{i+1} and D_{i+1}, and that we are only adding new information about the domain, i.e. $D_{i+1} \models D_i$. Then,*

$$\underbrace{(R_{i+1} \cup D_{i+1}) \not\models \bot}_{consistency} \wedge \underbrace{(R_{i+1} \cup D_{i+1}) \models R_i}_{completeness\ w.r.t.\ R_i} \implies (R_{i+1} \cup D_{i+1}) \models R_i \cup D_i$$

In other words, if we can prove that:

cons) *our new requirements are consistent with the domain (i.e., they are not asking for something that is impossible in the real world), and that*

$compl_{md}$) *the new requirements and domain description, together, do not contradict the previous requirements,*

then $(R_{i+1} \cup D_{i+1}) \models R_i \cup D_i$ holds.

Notice that we are not asking for the requirements themselves to be monotonic, i.e., it can well be that $R_{i+1} \not\models R_i$.

A similar lemma can be proved for the case in which the domain refinement is not monotonic, while requirements evolution is:

Lemma 2 (Monotonic requirements refinement) *Let us assume that we are performing an evolution step, from R_i and D_i to the subsequent versions*

R_{i+1} and D_{i+1}, and that we are only adding new requirements, i.e. $R_{i+1} \models R_i$. Then,

$$\underbrace{(R_{i+1} \cup D_{i+1}) \not\models \bot}_{consistency} \wedge \underbrace{(R_{i+1} \cup D_{i+1}) \models D_i}_{completeness\ w.r.t.\ D_i} \implies (R_{i+1} \cup D_{i+1}) \models R_i \cup D_i$$

In other words, if we can prove that:

cons) *our new requirements are consistent with the domain described so far, and that*

$compl_{mr}$) *the new requirements and domain description, together, do not contradict the previous domain description,*

then $(R_{i+1} \cup D_{i+1}) \models R_i \cup D_i$ holds.

Finally, we have the case in which both the requirements and the domain change in a non-monotonic way. Such an evolution step amounts to essentially changing everything we have collected so far in our analysis in an arbitrary fashion (e.g., we could scrap both R_i and D_i and take completely new R_{i+1} and D_{i+1}). In this case, the properties we are interested in must be proved directly, i.e.

cons) $(R_{i+1} \cup D_{i+1}) \not\models \bot$, and

$compl_{nm}$) $(R_{i+1} \cup D_{i+1}) \models R_i \cup D_i$.

An important result stemming from the considerations above is expressed by the following

Theorem 1 (Inductive correctness) *Let $(R_0, D_0), \ldots, (R_{n+1}, D_{n+1})$ be a chain of evolution steps in the development of a specification. If at each step the appropriate proof obligations cons, $compl_{md}$, $compl_{mr}$, $compl_{nm}$ are discharged, according to the lemmas above, then*

$$\forall i \in [0..n], (R_{i+1} \cup D_{i+1}) \models (R_i \cup D_i)$$

It follows by simple induction that

$$(R_{n+1} \cup D_{n+1}) \models (R_0 \cup D_0)$$

or, by reverting to our original names and remembering that $D_0 = \emptyset$,

$$(S \cup D_{n+1}) \models B$$

This last result is particularly relevant. In fact, the theorem states that, if at each step we discharge the appropriate proof obligations (summarized in

Condition	Kind of step	Proof obligations
$D_{i+1} \models D_i$	Monotonic domain evolution	*cons*, $compl_{md}$
$R_{i+1} \models R_i$	Monotonic requirements evolution	*cons*, $compl_{mr}$
none	Non-monotonic evolution	*cons*, $compl_{nm}$

Table 1
Proof obligations for different kind of evolution steps.

Table 1), we are assured that 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 (C2) of correctness.

The implications of Theorem 1 are twofold. On one side, the theorem links the two definitions of correctness that we presented in Section 1, showing that the customer- and business-oriented view of correctness in (C2) is just a coarse-grained view of the more formally defined notion of consistency+completeness given in definition (C1). On the other side, our framework provides guidance on how a potentially huge, complex, and cumbersome proof of correctness of a whole specification (i.e., $S \cup D \not\models \perp$ and $S \cup D \models B$) can be broken into smaller, simpler, step-wise proofs at each evolution step. This is useful both from a practical point of view (as will be discussed in Section 5) and from a methodological point of view.

3.4 Handling failed proofs and non-refining evolution

In real life, it is hardly ever the case that a development process consists only of correct refinement steps. Rather, back tracking, experimentation, sudden reversing of direction, and rework are the norm. It is thus important to be prepared to handle those cases in which the proof obligations shown above cannot be discharged (simply because the corresponding properties do not hold).

If the evolution step $i + 1$ just performed does not preserve consistency, i.e. $R_{i+1} \cup D_{i+1} \models \perp$, the situation is inemendable: the requirements are asking for something that is impossible in the real world. Either actions must be taken in the real world to change it so that the requirements become feasible (and D_{i+1} is changed accordingly to describe the new domain), or the infeasible requirements must be discarded (removing them from R_{i+1}).

If, instead, the evolution step does not preserve completeness, i.e. $(R_{i+1} \cup D_{i+1}) \not\models (R_i \cup D_i)$, several options are available. If something is missing from R_{i+1} (resp. D_{i+1}), the relevant information can be copied from R_i (resp. D_i) and added to the new set. If $R_{i+1} \cup D_{i+1}$ entails something that is in contradiction with the contents of $R_i \cup D_i$, the offending requirements or domain

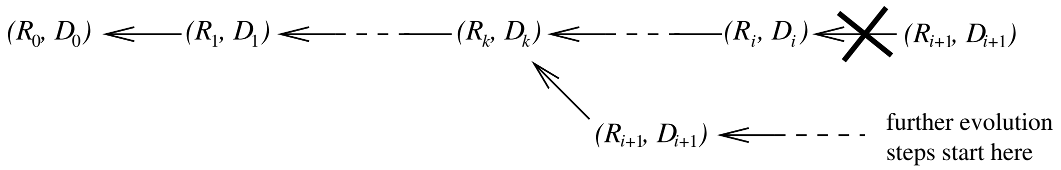

Fig. 3. Minimizing rework in case of failed completeness proof. The arrows represented completeness proofs; the crossed arrow represents the failed proof.

description statements can be retracted from the new set (so that information inconsistent with $R_i \cup D_i$ is no longer entailed), or from the previous set. In the latter case, the completeness of $R_i \cup D_i$ w.r.t. $R_{i-1} \cup D_{i-1}$ must be checked again.

Of course, adding unnecessary requirements (resp. domain description statements) or discarding necessary ones is not a solution if the offending change was brought about deliberately. In this case, we are explicitly stating that we do not want to maintain correctness w.r.t the previous evolution step because that step was in some way *wrong* and we want to reject it (as a kind of failed experiment). Still, we want to maintain correctness w.r.t. B. The most economic way to maintain it is to find the closest preceding step which allows us to maintain completeness, i.e. step k such that

$$(R_{i+1} \cup D_{i+1}) \models (R_k \cup D_k)$$

and

$$\forall j > k, (R_{i+1} \cup D_{i+1}) \not\models (R_j \cup D_j)$$

In most cases, a few checks (starting at step $i-1$ and going backward) will be sufficient to find a suitable correct ancestor. This approach allows us to start a new branch of correct development with no need to repeat all the proofs up to B, thus minimizing rework. The final result is depicted graphically in Figure 3. Notice that this technique extends naturally to the case in which $k = 0$, i.e., the whole requirements analysis process is started from scratch.

Finally, if no such k exists, and still we believe that our latest requirements and domain model are the “right” ones, then we have a fundamental clash between B and R_{i+1}, D_{i+1}. This can happen, for example, when the requirements analysis process reveals that the initial statement of business needs was inadequate, and did not represent faithfully the customer’s intentions, or that it was infeasible (given that in the course of the analysis we have gained more domain knowledge than was initially available). In this case, the only course left is to amend B so that $R_{i+1} \cup D_{i+1} \models B$. This, too, can be obtained in an incremental fashion, re-checking the correctness proofs working forward until the smallest k' is found such that $(R_{k'} \cup D_{k'}) \models B$. This last case completes our analysis.

4 A complete example

Let us illustrate the concepts presented above by means of a case study. We assume the role of a requirements engineer, and show how the requirements analysis for a simple control system — an automatic gate to regulate access to a building — can be performed in a rigorous way according to our framework. We also show a purely monotonic extension to our first implementation, and a case of non-monotonic extension (i.e., our customer changes his or her mind about the business goals), leading to some rework.

4.1 Problem statement

The business problem we are asked to solve can be stated as follows. Our customer, the security officer of a company, wants to install an automatic gate at the (only) entrance of a building, so that only authorized employees can access the building. We thus have :

$$B = \{\textsf{Only authorized persons can enter the building.}\}$$

From these business needs we start our investigation on how to turn B into a specification for a software system. Once again, notice that we are not questioning whether B is the "right" statement of business needs — we simply assume that our customer is in good faith when stating it. Of course, B *can* change during the development process, and we will show in Sections 4.3 and 4.4 how to handle such changes.

4.2 Requirements and domain evolution

At the beginning of the requirements analysis process, we do not have any specific knowledge about the domain, and our initial requirement is given by B alone. In other words,

$$\begin{aligned} B = R_0 &= \{\textsf{Only authorized persons can enter the building.}\} \\ D_0 &= \{\} \end{aligned}$$

R_0 cannot be directly implemented, thus we need to refine it. First, we perform a simple formal transformation of the requirement to simplify it:

$$\begin{aligned} R_1 &= \{\text{Authorized persons can enter the building. Non-authorized persons are prevented from entering the building.}\} \\ D_1 &= \{\} \end{aligned}$$

Since clearly $D_1 \models D_0$, we must prove only *cons* ($R_1 \cup D_1 \not\models \perp$) and $compl_{md}$ ($R_1 \cup D_1 \models D_0$) — both of which are trivial. Hence, our first step is correct (in sense (C1)). For the next step we need to know (a) how to distinguish authorized from non-authorized persons, and (b) how to allow or deny access to the building. Regarding (a), we investigate company procedures and find out that each employee is issued an identity badge that can also act as a swipe card. Each card carries a Unique ID number (UID) that is uniquely associated with the person to which the card is issued.[5] Regarding (b), we decide that a locked bullet-proof glass door (gate) is an adequate means to prevent access to the building. These information are now part of our domain knowledge:

$$\begin{aligned} R_2 &= \{\text{Authorized persons can enter the building. Non-authorized persons are prevented from entering the building.}\} \\ D_2 &= \{\text{Authorized persons have ID cards whose UID is authorized. An open gate allows entering the building. A locked gate prevents entering the building.}\} \end{aligned}$$

This time, $R_2 \models R_1$, hence we have to prove *cons* and $compl_{mr}$ ($R_2 \cup D_2 \models D_1$), which again hold trivially. We can now perform a more substantial step: we decide that we want to recognize people by their ID cards, and update our requirements accordingly:

$$\begin{aligned} R_3 &= \{\text{Persons who have an ID card whose UID is authorized can enter the building. Persons who do not have an ID card whose UID is authorized are prevented from entering the building.}\} \\ D_3 &= \{\text{Authorized persons have ID cards whose UID is authorized. An open gate allows entering the building. A locked gate prevents entering the building.}\} \end{aligned}$$

The relevant proof obligations are *cons* and $compl_{md}$. The latter can be discharged by substituting the definition for "authorized" given in D_3 into R_3 (the result is indeed R_2, thus proving that $R_3 \cup D_3 \models R_2$), while the former is trivial.

To make implementation of R_3 possible, we must be able to discern whether a

[5] This procedure is not very robust, since a badge could be stolen and used by an intruder to impersonate a legitimate employee. However, this weakness stems from the domain itself, i.e., from company policies, and not from the software to be developed.

person has an ID card or not, and need a way to know if a given UID number is authorized or not. We can ask a person to swipe his or her ID card through a swipe card reader to prove ownership of an ID card. The reader will provide the UID number, that we can then match against a database of authorized UIDs (AuthDB for short). These facts must be recorded among our domain knowledge,

R_4 = {Persons who have an ID card whose UID is authorized can enter the building. Persons who do not have an ID card whose UID is authorized are prevented from entering the building.}

D_4 = $D_3 \cup$ {Persons who swipe an ID card, have an ID card. If a person swipes an ID card, the card reader provides the UID of the card. The UID of a card is authorized if it appears in AuthDB.}

We omit the correctness proof for this step, that is once again simple enough, and move on to a more substantial refinement: we refine the notion of "possessing an authorized ID card" to the specific sequence of actions and tests that must be conducted:

R_5 = {When a person swipes an ID card, if the UID provided by the card reader appears in AuthDB, then the gate must be opened. If a person does not swipe an ID card, or the UID provided by the card reader does not appears in AuthDB, then the gate must be locked.}

D_5 = D_4

Checking the correctness of this last step is a slightly more complex that in the previous cases. We must prove *cons* and $compl_{md}$. The former is trivial, but the latter requires some attention. Formally, our proof obligation is $R_5 \cup D_5 \models R_4$. While the reasoning could be carried on informally, we choose this time to resort to formal logic for added confidence (and presentation economy). Unfortunately, as shown in Appendix B (to which we refer the reader for the details of the proof), it turns out that $compl_{md}$ cannot be proved in this case — and, indeed, the property does not hold.

The reason for this failure lies in the fact that R_4 asks for authorized persons — i.e., according to our domain, persons who *possess* an ID card whose UID is authorized — to be granted access, while R_5 restricts access to those authorized persons that, additionally, take the extra step of *swiping* the card in the card reader. Thus, the combination of R_5 and D_5 does not provide all the functionalities requested by R_4: we are being incomplete w.r.t. R_4 (and, by Theorem 1, also w.r.t. B).

Faced with this incompleteness, we can try to solve it by relaxing our previous requirements (i.e., by dropping the unreasonable assumption that the mere possession of an authorized card entitles the owner to access the building), or by introducing more sophisticated technology in the domain (i.e., using Bluetooth-based cards and sensors, so that possessor of a card can be detected via radio signals), or by explicitly stating a distrust assumption (i.e., stating that we assume that a person does not have a card unless he or she swipes it in the card reader).

We opt for the latter, thus obtaining

$R'_5 = R_5$ = {When a person swipes an ID card, if the UID provided by the card reader appears in AuthDB, then the gate must be open. If a person does not swipe an ID card, or the UID provided by the card reader does not appears in AuthDB, then the gate must be locked.}

D'_5 = $D_4 \cup$ {A person is presumed not to have a card unless he or she swipes it in the card reader.}

Both *cons* and $compl_{md}$ hold for this set, thus proving that a system respecting R'_5, deployed in the domain described by D'_5, would be correct with respect to our customer's business needs in B.

The evolution process can continue in a similar vein, until the requirements are directly implementable on available hardware. In our example, we still have to specify, in D, how UIDs can be read from the card reader, how the gate can be locked or opened, and how can the system query the AuthDB. The final R could, for example, be similar to

R_{final} = { When the system receives a "card passed" signal from the reader, it reads the UID from the reader buffer. If that UID appears in AuthDB, the system sends an "unlock command" to the gate controller. After 5 seconds, it sends a "lock command" to the gate controller. If, after 10 more seconds, the gate sensor still reports "gate open", the system sounds the warning alarm, until the sensor reports "gate closed". . . . }

with a correspondingly detailed D_{final}. It is interesting to note that, although our R_{final} and D_{final} would be consistent and complete w.r.t. B (if we have discharged all our incremental proof obligations), they may well implement functionalities that were not asked for originally. In R_{final} above, sounding a warning alarm if the gate is stuck for some mechanical reason was not requested in B. In fact, our definition of correctness does not include any notion of *minimality* — another useful characteristic of its own, but outside the scope of this work.

4.3 A monotonic extension

We consider now the case in which business needs change during the requirements analysis process. Let us suppose that our customer realizes that logging of accesses is also needed, thus re-defining the business needs as

B' = {Only authorized persons can enter the building. The time and identity of every person entering the building must be logged.}

How can we introduce new functionalities in our requirements to satisfy the new business need, while at the same time guaranteeing correctness and minimizing rework? As suggested in Section 3.4, an incremental approach can be used.

Let us for simplicity assume that the change in B happens while we are at the fifth step of evolution, i.e. R'_5 and D'_5 from the previous section. By checking our requirements and domain against the new business needs, we discover that we are no longer complete w.r.t. B'. In fact, our old requirements lack the logging functionality altogether. To solve the problem, we can simply try to add new requirements to R'_5, obtaining, for example,

R_6 = {When a person swipes an ID card, if the UID provided by the card reader appears in AuthDB, then the gate must be opened, and the UID and the current time are logged. If a person does not swipe an ID card, or the UID provided by the card reader does not appear in AuthDB, then the gate must be locked.}

D_6 = $D'_5 \cup$ {The identity of a person is given by the UID of his or her card.}

If we can prove that $R_6 \cup D_6 \not\models \perp$ and that $R_6 \cup D_6 \models B'$, we can discard the intermediate steps $R_1, D_1 \ldots R'_5, D'_5$ and continue development from R_6, D_6. However, this usually happens only when the business needs and the new requirements are not too far apart, i.e., if only a few evolution steps are being shortcutted. This is, fortunately, the case of our example: we are considering a *monotonic* extension of the business needs. By using the same techniques described in Appendix B, we can easily prove that $R_6 \cup D_6$ is consistent and complete with respect to B', with no need for rework.

Sometimes, the added functionality cannot be easily proved to completely satisfy the new business needs. In this case, more abstract versions of the new requirements should be added to previous versions of the requirements document. In our example, we could add the new functionality to R'_5 (let us call the new version R''_5), and prove $R_6 \cup D_6 \models R''_5$, $R''_5 \cup D'_5 \not\models \perp$, thus reducing the original problem to proving that $R''_5 \cup D'_5 \models B'$ (that can be solved inductively by the same technique).

4.4 A non-monotonic extension

As a final point in our example, we consider a non-monotonic extension to our customer's business needs, like the following one:

B'' = {Only authorized persons can enter the building, unless an emergency is detected — in which case, anyone can enter the building.}

As can be expected, our requirements (while still consistent) are now no longer complete: handling of emergencies is missing altogether. Here again we could simply add the new functionality to R'_5, but if we try to discharge our proof obligations, we discover that by so doing our requirements are complete w.r.t. B'', but no longer consistent. This is the typical case in which consistency and completeness are at odds with each other. In fact, what we have is

R'_6 = {On emergencies, the gate must be opened. When a person swipes an ID card, if the UID provided by the card reader appears in AuthDB, then the gate must be open. If a person does not swipe an ID card, or the UID provided by the card reader does not appears in AuthDB, then the gate must be locked.}

$D'_6 = D'_5$

As can be noted, in case of an emergency the first requirement states that the gate must be open, while the third requirement states that it must be locked (unless someone authorized is entering at the same time). In this simple case, consistency can be restored by appending an extra condition to the third requirement: "…then the gate must be locked, unless there is an emergency". This is sufficient to restore consistency, while maintaining completeness. Indeed, this last version of our requirements in correct w.r.t. B''.

4.5 Discussion

We can now look back at the question we posed in the introduction. As the discussion in Section 3 and the case study have shown, we believe that the two notions of correctness we mentioned in Section 1 are indeed closely related to each other. It is thus important to explore the more formal treatment implied by definition (C1) (consistency+completeness), because that provides us with the pragmatically more relevant correctness in (C2) (satisfaction of business goals). We can state that a formal treatment of consistency, completeness and correctness at each evolution step in a requirements specification process do actually allow us to satisfy the business goals of the customer (beside being a valuable contribution by itself).

Indeed, at the end of our example, having discharged all the relevant proof obligations at each step, we can rest assured that our R_{final} and D_{final} — detailing the behavior of the system in terms of signals received from input ports and of commands sent to output ports[6], and thus directly implementable on a standard machine — do actually satisfy the business needs "only authorized persons can enter the building". The same assurance would be much more difficult to obtain without a rigorous framework that clearly and rationally states which properties must be proved, indicating how to structure the proofs into manageable pieces.

5 The three Cs in practice

The framework we have proposed has a number of interesting connections with other widely studied problems in requirements engineering. In this section we discuss how our framework relates to other topics of interest of practitioners and researchers alike.

5.1 Formality and informality

To perform consistency, completeness and correctness checking effectively and to be able to automate this process (in order to assist the requirements engineers in some of their more difficult and mundane tasks), the specification has to be expressed in a formal notation. This is because computer-based analysis requires an explicit formal semantics which provides the basis for the algorithms that carry out the analysis. This is precisely the approach that has been taken by proponents of formal methods in RE. Indeed much of the RE research effort over the last three decades has been concentrated on developing new formal requirements specification languages so that tasks such as syntax correctness, reasoning about requirements and checking their consistency can be automated in ways that are similar to how programs are compiled and managed.

[6] Note that this level of description lends itself naturally to specification and further analysis through established formal methods, e.g. SCR and related tools. This is an important feature of our framework, that provides integration with other approaches and guidance for those high-level activities that are normally left out of "hard" formal methods.

Although it is an advantage to have a formal proof of correctness of a specification (e.g. as in [38]), 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, as we did in the initial stages of our case study. 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.

5.2 Tolerating inconsistency and incompleteness

It is important to stress that our framework identifies which kind of consistency and completeness checks must be performed to verify correctness, but do not prescribe how to handle any error found, 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 proof obligations we discussed in Section 3 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 [2,3,9,39,40]), if they reflect a genuine conflict of goals or simply there is not enough information available yet to decide how to correct them. Alternatively, they can be 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.

5.3 Correctness measures and prioritization

The desire of being able to tolerate inconsistency and incompleteness also introduces the need to *measure* the degree of consistency and completeness that has been attained. Moreover, a priority could be assigned to requirements (and domain description statements), so that in case of incorrectness, the less important requirements can be discarded, while retaining more important ones. 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.

As a measure for the "degree of consistency" we consider the ratio between the size of a maximal consistent subset (mcs) of $R \cup D$ and the size of the whole set[7], i.e.:

$$\delta_{cons}(R, D) = \frac{|mcs(R \cup D)|}{|R \cup D|}$$

For consistent R and D, $\delta_{cons}(R, D) = 1$, whereas the measure tends to 0 for increasing degree of inconsistency. Analogously, as a measure for the "degree of completeness" of a set S w.r.t. $R \cup D$, we consider the ratio between the size of a maximal subset of S that is entailed by $R \cup D$ (maximal entailed subset, or mes) and the size of the whole set S, i.e.:

$$\delta_{compl}(R, D, S) = \frac{|mes(R \cup D, S)|}{|S|}$$

This measure, too, has value 1 when completeness holds, and assumes progressively lower values, down to 0, for decreasing completeness. Notice that both measures indicate the degree of consistency and completeness in merely numeric terms, and say nothing about how relevant the *effect* of any inconsistency or incompleteness is on the overall satisfaction of the business goals.

At each step of evolution, it is important to identify any emerging incorrectness resulting from adding new requirements or domain description statements. By providing automated tools (e.g. [41], where inconsistency was addressed), that suggest alternative solutions on how to manage these problematic additions, together with the corresponding measures of completeness and consistency 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 — in requirements specifications.

5.4 Process issues

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

[7] Of course, there may be more than one maximal consistent subset: we consider here the size of any of the largest maximal consistent subset, thus our definition is sound. The same holds for the definition of δ_{compl} below.

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 properties described in Section 3) and automatically, by using appropriate tools, like the simple model checker SMACK we developed and used in the example (see Appendix B), or the more sophisticated tools described in [41,42]. 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 works 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 technique we have described can also be used to guide the requirements engineers and stakeholders in deciding when to stop eliciting more requirements, i.e. when the specification can be proved to be "sufficiently" correct with respect to the business needs, taking for example the measures above as an indicator. As part of future works we plan to perform an empirical study of informal correctness justifications as "stop" criteria for requirements analysis.

Traceability is also an important property that needs to be addressed within the requirements evolution process. Because software engineers use traceability to guarantee that the requirements specifications and implementation of subsystems are complete with respect to the specified overall system requirements and design constraints, any specification methodology that supports evolution must have effective support for traceability. Our approach introduces a new kind of relationships among requirements and domain description statements that can be profitably traced, namely *support* (the tracing can be automatic for formal requirements). The *support* relationship relates a requirement or domain description statement $s \in R_i \cup D_i$ to all those requirements and domain description statements at step $i+1$ that partake in the entailment proof that guarantees the completeness of $R_{i+1} \cup D_{i+1}$ with respect to s. Thus, our approach provides "for free" a particularly strong dependency relationship that can be used for effective tracing of requirements and change impact analysis.

5.5 RE education and training

The process of managing the interplay between the three Cs of requirements discussed in this paper can be used as a pedagogical tool to assist the RE students develop a fuller understanding of what quality factors to look for during the evolution of requirements. This process gives students a fuller appreciation of the range of issues involved in requirements evolution and change management. Furthermore, by demonstrating how inconsistency or incompleteness could easily be introduced in the specification that ultimately threatens the correctness, students will learn to be more careful and cautious when expressing requirements and domain descriptions. In our experience, developing a thorough understanding of these important issues is fundamental in RE education (see for example [43]).

6 Conclusions 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 framework is a starting point for a 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 scientists... 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 [1] 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 [41] as well as in formal logic [44], in both cases applying a form of non-monotonic logic to carry out consistency proofs. 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 3. 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. Also, support for metrication, prioritization, traceability, and stop criteria will be implemented in tools, providing the requirements engineer with a complete support environment for the delicate requirements evolution process.

References

[1] M. Jackson, Software Requirements & Specifications: a lexicon of practice, principles and prejudices, Addison Wesley, Great Britain, 1995.

[2] B. Nuseibeh, To be and not to be: on managing inconsistency in software development, in: Proceedings of the Eight IEEE International Workshop on Software Specifications and Design (IWSSD'96), IEEE Computer Society Press, 1996, pp. 164–169.

[3] S. Easterbrook, B. Nuseibeh, Managing inconsistencies in an evolving specification, in: Proceedings of the Second International Symposium on Requirements Engineering (RE95), York, England, 1995, pp. 48–55.

[4] S. M. Easterbrook, E. E. Beck, J. S. Goodlet, L. Plowman, M. Sharples, C. C. Wood, A survey of empirical studies of conflict, in: S. M. Easterbrook (Ed.), CSCW: Cooperation or Conflict?, Springer-Verlag, London, 1993, pp. 1–68.

[5] W. Robinson, S. Volkov, A meta-model for restructuring stakeholders requirements, in: Proceedings of the International Conference on Software Engineering (ICSE97), Boston, USA, 1997, pp. 140–149.

[6] A. van Lamsweerde, R. Darimont, E. Letier, Managing conflicts in goal-driven requirements engineering, IEEE Transactions on Software Engineering: special issue on Managing Inconsistency in Software Development 24 (11) (1998) 908–926.

[7] G. Cugola, E. D. Nitto, A. Fuggetta, C. Ghezzi, A framework for formalizing inconsistencies and deviations in human-centred systems, ACM Transactions on Software Engineering and Methodology 5 (3) (1996) 191–230.

[8] T. M. Hagensen, B. B. Kristensen, Consistency in software system development: Framework, model, techniques, and tools, in: Software Engineering Notes (Proceedings of ACM SIFSOFT Symposium on Software Development Environment), SIGSOFT and ACM Press, 1992, pp. 58–67, 17(5).

[9] R. Balzer, Tolerating inconsistency, in: Proceedings of the 13th IEEE International Conference on Sofware Engineering (ICSE13), IEEE Computer Society Press, Austin, Texas, 1991, pp. 158–165.

[10] A. Borgida, S. Greenspan, and J. Mylopoulos, Knowledge representation as the basis for requirements specifications, IEEE Computer April (1985) 82–91.

[11] D. W. Cordes, D. L. Carver, Evaluation methods for user requirements documents, Information and System Technology 31 (4) (1989) 181–188.

[12] E. Dubois, P. Du Bois, and A. Rifaut, Elaborating, structuring and expressing formal requirements of composite systems, in: Proceedings of CAiSE, Springer-Verlag, Berlin, 1992, pp. 327–347.

[13] S. Easterbrook, B. Nuseibeh, Using viewpoints for inconsistency management, IEE Software Engineering Journal 11 (1) (1996) 31–43.

[14] C. L. Heitmeyer, R. D. Jeffords, B. G. Labaw, Automated consistency checking of requirements specifications, ACM Transactions on Software Engineering and Methodology 5 (3) (1996) 231–261.

[15] B. Nuseibeh, J. Kramer, and A. Finkelstein, A framework for expressing the relationships between multiple views in requirements specification, IEEE Transactions on Software Engineering 20 (10) (1994) 760–773.

[16] H. B. Reubenstein and R. C. Waters, The requirements apprentice: Automated assistance for requirements acquisition, IEEE Transactions on Software Engineering 17 (3) (1991) 226–240.

[17] I. Sommerville, P. Sawyer, S. Viller, Viewpoints for requirements elicitation: A practical approach, in: Proceedings of the IEEE 3rd International Conference on Requirements Engineering, (ICRE98), IEEE Computer Society Press, Colorado springs, USA, 1998, pp. 74–81.

[18] R. Kowalski, Logic for Problem Solving, North Holland Elsevier, New York, 1979.

[19] R. R. Lutz, Analyzing software requirements errors in safety-critical, embedded systems, in: Proceedings of the First IEEE International Symposium on Requirements Engineering (RE93), 1993, pp. 35–46.

[20] A. M. Davis, Software Requirements: Analysis and Specification, 2nd Edition, Prentice Hall, 1993.

[21] B. W. Boehm, Verifying and validating software requirements and design specifications, IEEE Software 1 (1) (1984) 75–88.

[22] N. Leveson, Completeness in formal specification language design for process-control systems, in: Proceedings of the Third Workshop on Formal Methods in Software Practice, Portland, Oregon, 2000, pp. 75–87.

[23] M. S. Jaffe, N. G. Leveson, M. P. E. Heimdahl, B. E. Melhart, Software requirements analysis for real-time process-control systems, IEEE Transactions on Software Engineering 17 (3) (1991) 241–258.

[24] E. Letier, A. van Lamsweerde, Requirements analysis: Deriving operational software specifications from system goals, in: Proceedings of the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering, ACM Press, Charleston, South Carolina, 2002, pp. 119–128.

[25] J. Mylopoulos, L. Chung, E. Yu, From object-oriented to goal-oriented requirements analysis, Commun. ACM 42 (1) (1999) 31–37.

[26] H. Kaiya, H. Horai, M. Saeki, AGORA: Attributed goal-oriented requirements analysis method, in: Proceedings of the Tenth IEEE Joint International Requirements Engineering Conference (RE02), Essen, Germany, 2002, pp. 13–22.

[27] M. Woodside, D. Petriu, K. Siddiqui, Performance-related completions for software specifications, in: Proceedings of the 24th IEEE International Conference on Software Engineering (ICSE'02), ACM Press, Orlando, Florida, 2002, pp. 22–32.

[28] A. Bouhoula, Simultaneous checking of completeness and ground confluence, in: Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE'00), Grenoble, France, 2000, pp. 143–153.

[29] F. T. Sheldon, H. Y. Kim, Z. Zhou, A case study: validation of guidance control software requirements for completeness, consistency and fault tolerance, in: Proceedings of the 2001 Pacific Rim International Symposium on Dependable Computing (PRDC 2001), Seoul, Korea, 2001, pp. 311–318.

[30] K. Yue, What does it mean to say that a specification is complete?, in: Proceedings of the IEEE International Workshop on Software Specifications and Design (IWSSD'87), 1987, pp. 42–49.

[31] C. M. Lott, Correctness is congruent with quality, ACM Software Engineering Notes 15 (5).

[32] M. Jackson, The world and the machine, in: Proceedings of 17th IEEE International Conference on Software Engineering (ICSE17), IEEE Computer Society Press, Seattle, USA, 1995, pp. 283–292.

[33] P. Zave, M. Jackson, Four dark corners of requirements engineering, ACM Transactions on Software Engineering and Methodology 6 (1) (1997) 1–30.

[34] M. Jackson, P. Zave, Domain descriptions, in: Proceedings of the First IEEE International Symposium on Requirements Engineering (RE93), IEEE Computer Society Press, 1993, pp. 56–64.

[35] D. Bjorner, C. W. George, B. S. Hansen, H. Laustrup, S. Prehn, Models of railway systems infrastructure, in: Proceedings of the First Workshop on Formal Methods in Railway Industry, Nieuwegein, Netherlands, 1997, http://www.ifad.dk/Projects/FMERail/proceedings1.html.

[36] M. Jackson, Problem Frames: Analyzing and structuring software development problems, Addison Wesley, Great Britain, 2001.

[37] H. Katsuno, A. O. Mendelzon, On the difference between updating a knowledge base and revising it, in: J. F. Allen, R. Fikes, E. Sandewall (Eds.), Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning (KR'91), Morgan Kaufmann, San Mateo, California, 1991, pp. 387–394.

[38] M. P. E. Heimdahl, N. G. Leveson, Completeness and consistency in hierarchical state-based requirements, IEEE Transactions on Software Engineering 22 (6) (1996) 363–377.

[39] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, B. Nuseibeh, Inconsistency handling in multi-perspective specifications, IEEE Transactions on Software Engineering 20 (8) (1994) 569–577.

[40] A. Hunter, B. Nuseibeh, Managing inconsistent specifications: Reasoning, analysis and action, ACM Transactions on Software Engineering and Methodology 7 (4) (1998) 335–367.

[41] D. Zowghi, V. Gervasi, A. McRae, Using default reasoning to discover inconsistencies in natural language requirements, in: Proceedings of the Eight Asia-Pacific Software Engineering Conference (APSEC'01), Macau, China, 2001, pp. 133–140.

[42] V. Gervasi, B. Nuseibeh, Lightweight validation of natural language requirements, Software: Practice & Experience 32 (2) (2002) 113–133.

[43] D. Zowghi, S. Paryani, Teaching requirements engineering through role playing, submitted to the 11th IEEE Joint International Requirements Engineering Conference (RE03).

[44] D. Zowghi, R. J. Offen, A logical framework for modelling and reasoning about the evolution of requirements, in: Proceedings of the Third IEEE International Symposium on Requirements Engineering (RE97), 1997, pp. 247–259.

Appendices

A Notation

Throughout the paper, we use standard notation from set theory and logic. In particular, we use capital letters (A-Z, possibly subscripted) to indicate sets of *statements*, in various roles: as business needs, requirements, domain descriptions, or specifications. The *cardinality* of such a set A, i.e., the number of statements it contains, is indicated by $|A|$. Set union $\cup$ and set intersection $\cap$ maintain their usual meaning.

The entailment operator $\models$ from logic is used to indicate derivation of a conclusion c from a set of premises P, according to a given set of derivation rules. In this case, we write $P \models c$. Entailment is naturally extended to a set of conclusions C, denoted as $P \models C$. The negation $P \not\models c$ indicates that from the premises in P it is not possible to ensure that c holds.

Notice that in the paper we do not assume a specific set of derivation rules: for our purposes, first-order logic is as acceptable as is reasonable inference on English statements (see Section 5.1 for details).

We use the symbols $\wedge$ (*and*), $\vee$ (*or*), $\neg$ (*not*), $\implies$ (*implication*), $\forall$ (*for all*), and $\exists$ (*exists*) in their standard meaning. The symbol $\perp$ (*bottom*) indicates the tautologically false statement, i.e., logic impossibility. $\perp$ can never hold, thus it can only be derived from an *inconsistent* set of statements, i.e., a set of statements that themselves can never hold all at the same time. Hence, $A \models \perp$ means that A is inconsistent. Conversely, $A \not\models \perp$ means that A is consistent.

B The completeness proof for Section 4.2

In Section 4.2 we had to prove the correctness of the fifth step w.r.t. the fourth step, i.e.

$$R_5 \cup D_5 \not\models \perp \quad \text{(consistency)}$$

$$R_5 \cup D_5 \models R_4 \quad \text{(completeness)}$$

In previous steps, the proofs have been simple enough that they could be carried on just by inspection of the requirements. This time, however, the proof is more substantial, and we choose to use a propositional logic based form for added confidence. Table B.1 shows the correspondence between the English form and the logic form of our requirements and domain model.

We carry on the proof using a simple ad-hoc model checker for propositional logic, called SMACK, that we developed in-house in support of our framework. SMACK input consists of a number of set assignments of the form

S = { p_1. ... p_n. }

where the p_i are propositional logic formulae (e.g., those in Table B.1), and of a number of completeness or consistency properties to prove, of the form

```
R U D |= S
```

or

cons `R U D`

where `R`, `D` and `S` are sets defined as shown above. Given this input, SMACK generates, compiles, and executes a C program that performs an exhaustive search for violations of the properties listed in the input.

If we provide SMACK with the definition of the three sets R_5, D_5, and R_4, consistency is easily verified. But if we ask to prove $R_5 \cup D_5 \models R_4$, the proof fails, returning the following counterexample:

```
has(p,card) ∧ hasUID(card,UID) ∧ inAuthDB(UID) ∧ authorized(UID)
∧ authorized(p) ∧ ¬swipes(p,card) ∧ ¬canEnter(p) ∧ ¬openGate ∧
lockedGate
```

That is: we have a person who has a card; the UID of the card appears in AuthDB; thus, the card is authorized and as a consequence the person is authorized to enter the building. However, as long as the person does not swipes the card, she cannot enter: the gate is locked. This is of course in contrast with R_4, that states that *authorized* persons must be granted access, where authorization is in fact the mere *possession* of an authorized card. There is no mention of any need to perform special actions (i.e., swiping the cards) in R_4. Thus, in our fifth refinement we are failing to *completely* satisfy what is requested by R_4 (and, by Theorem 1, we are not correct with respect to B).

It is worthwhile to remark that the problem could be "fixed" in various ways, but not all of them are significant in terms of requirements evolution. For example, by writing `swipes(p,card)` $\iff$ `has(p,card)`, $compl_{md}$ holds, and the proof succeeds, but we are describing an odd domain, where: (a) if a person swipes a card, then he or she has the card, but also (b) if a person has a card, he or she is supposed to swipe it in the card reader all the time (whether the person wants to access the building or not). Notice that this is certainly not what is desired, as hinted by optionality in "Only authorized persons *can* enter the building."

A better solution is to refine our definition of an authorized person as follows:

> Authorized persons are those that swipe ID cards whose UID is authorized.

that in terms of propositional logic amounts to writing

$$\texttt{swipes(p,card)} \land \texttt{hasUID(card,UID)} \land \texttt{authorized(UID)} \iff \texttt{authorized(p)}$$

but this, too, is not entirely satisfactory: in the customer's view of the world, a person is authorized after the corresponding card has been issued, as long as its UID code is found in AuthDB. Thus, 'being authorized' is a permanent property of a person, and not a property that holds for a short moment while the person swipes the card, and disappears immediately afterwards.

The best solution is probably to introduce an explicit expression of mistrust in the domain:

> A person is presumed not to have a card unless she swipes it in the card reader.

that is,

$$\neg\texttt{swipes(p,card)} \implies \neg\texttt{has(p,card)}$$

Notice that, albeit in logical terms the conjunction of this last statement with the previously existing $\texttt{swipes(p,card)} \implies \texttt{has(p,card)}$ is equivalent with the $\texttt{swipes(p,card)} \iff \texttt{has(p,card)}$ we rejected above, its meaning in terms of the interpretation of the requirements is much clearer, and thus preferable. With this last change, both completeness and consistency hold; we can proceed with the requirements analysis, reassured that our evolution step is correct.

Requirements R_5	
When a person swipes an ID card, if the UID provided by the card reader appears in AuthDB, then the gate must be open.	$\texttt{swipes(p,card)} \land \texttt{read(UID)} \land \texttt{inAuthDB(UID)} \implies \texttt{openGate}$
If a person does not swipe an ID card, or the UID provided by the card reader does not appears in AuthDB, then the gate must be locked.	$\neg\texttt{swipes(p,card)} \lor (\texttt{read(UID)} \land \neg\texttt{inAuthDB(UID)}) \implies \texttt{lockedGate}$
Domain model D_5	
Authorized persons have ID cards whose UID is authorized.	$\texttt{has(p,card)} \land \texttt{hasUID(card,UID)} \land \texttt{authorized(UID)} \iff \texttt{authorized(p)}$
An open gate allows entering the building.	$\texttt{openGate} \implies \texttt{canEnter(p)}$
A locked gate prevents entering the building.	$\texttt{lockedGate} \implies \neg\texttt{canEnter(p)}$
Persons who swipe an ID card, have an ID card.	$\texttt{swipes(p,card)} \implies \texttt{has(p,card)}$
If a person swipes an ID card, the card reader provides the UID of the card.	$\texttt{swipes(p,card)} \land \texttt{hasUID(card,UID)} \implies \texttt{read(UID)}$
The UID of a card is authorized if it appears in AuthDB.	$\texttt{inAuthDB(UID)} \iff \texttt{authorized(UID)}$

Requirements R_4	
Persons who have an ID card whose UID is authorized can enter the building.	$\texttt{has(p,card)} \land \texttt{hasUID(card,UID)} \land \texttt{authorized(UID)} \implies \texttt{canEnter(p)}$
Persons who do not have an ID card whose UID is authorized are prevented from entering the building.	$\neg\texttt{has(p,card)} \lor (\texttt{hasUID(card,UID)} \land \neg\texttt{authorized(UID)}) \implies \neg\texttt{canEnter(p)}$

Table B.1
The R_5, D_5 and R_4 sets in English and in propositional logic.