Design Goals for a Progressive Formalization Language in Systems Requirements Engineering

Sami Dahoux

Abstract Systems engineering has progressively evolved towards model-based practices in order to manage the growing complexity of cyber-physical projects. In parallel, formal early verification and validation (V&V) techniques, including simulation, model checking, and static analysis, have reached industrial maturity. However, adoption of these practices in industrial organizations remains limited, in part due to a low perceived return on investment. A key contributing factor is the *authoring gap*: no specification language currently occupies the right level of abstraction to simultaneously serve engineers writing natural language requirements, teams operating model-based workflows, and formal V&V back-ends. In this paper, we establish a set of six named design goals for REQ, a general-purpose textual requirements specification language designed to close this gap through *progressive formalization*. Each goal is motivated by observed engineering practice, paired with a language decision and supported by a rationale. A set of edition tools presented in Section 4, supports the contribution.

1 Introduction

This section presents the context and motivating observations behind the proposed work, covering requirements engineering practice, model-based approaches, and formal verification, before identifying the authoring gap that REQ is designed to address.

Qitab, 17 Rue de Clignancourt, Paris, France e-mail: sami@qitab.io

1.1 Requirements Engineering

The systems engineering process, while encompassing activities well beyond requirements elicitation, analysis, and authoring, is most often driven and monitored through requirements documents. Requirements constitute the primary basis for decision-making and progress assessment throughout the development lifecycle [20]. Requirements management platforms, referred to as Application Lifecycle Management (ALM) solutions in this paper, such as IBM DOORS [2] or Siemens Polarion [3], provide means to collect, link, and track requirements across a program. These tools have become a standard component of systems engineering practice, whose normalization is codified in ISO/IEC 15288 [1] and further driven by increasingly stringent certification requirements, particularly in safety-critical domains.

1.2 Model-Based Systems Engineering

In response to the growing complexity of system lifecycle activities, Model-Based Systems Engineering (MBSE) has emerged as a family of methodologies and modeling languages providing structured representations of system architecture, components, and behaviors at appropriate levels of abstraction. MBSE approaches aim to decouple systems-level reasoning from domain-specific expert knowledge, enabling shared functional and non-functional understanding across heterogeneous teams. The most widely adopted modeling language in this space is SysML v2 [13], an evolution of the UML-derived SysML v1 standard. Its constructs have been progressively refined to offer a comprehensive toolbox for representing both structural and behavioral system properties, including executable semantics through KerML [14].

1.3 Formal Verification

For cost- or safety-critical systems, verification and validation activities have evolved to enable engineers to carry out system verification as early as possible in the development lifecycle, thereby reducing rework costs. Simulation, formal proof, and static analysis are the most prominent representatives of this class of techniques. Formal verification has historically required specialist expertise, owing to the complexity of tool setup and the domain-specific knowledge required to obtain sound results. The progressive maturation of simulation frameworks, model checkers, and static analyzers has demonstrated the robustness and coverage of these approaches across a wide range of verification scenarios.

1.4 The Authoring Gap

Despite the maturity of the tools described above, in contemporary industrial practice, requirements are most commonly recorded as free-form or semi-structured natural-language text in documents, ALM, and issue trackers, rather than in formal specification languages or models. Therefore, the lack of verifiable semantics. Links between requirements, modeling elements, and formal verification targets are established manually, and the consistency of these cross-artifact relationships must be re-evaluated at each development milestone.

Furthermore, natural language requirements frequently contain implicit formal expressions: constraints, physical quantities, timing conditions, glossary terms that are not machine-interpretable in their original form. As requirements evolve, these latent formal structures are lost, and the correspondence between documents, models, and verification artifacts degrades silently. This phenomenon, wherein the three artifact classes diverge under the pressure of real-world project evolution, is what we term the *authoring gap*. The consequence is a structural risk of specification drift, manifesting as ambiguity, omissions, and inconsistency across the engineering artifacts, as observed in practice [9]. This represents a significant risk to the outcome of a systems engineering program, particularly in contexts where formal evidence of correctness is required.

We observe that the perceived benefits of model-based and formal practices are frequently offset by the organizational complexity they introduce [12, 16]. The practices that can improve requirement quality, support shared understanding, and yield strong correctness guarantees simultaneously risk scattering knowledge across incompatible artifact formats, requiring additional integration effort from all stakeholders. A specification language bridging informal requirements and formal V&V back-ends, while remaining authorable by engineers without formal methods expertise, can substantially improve the return on investment of these practices by eliminating manual translation effort and maintaining a single authoritative artifact.

1.5 The REQ *Language*

The REQ language is a textual specification language designed to address the authoring gap described above. A full description of the language, its constructs, and its toolchain is beyond the scope of this paper. This paper focuses exclusively on the design rationale: the methodological principles and design goals that govern the language, independently of any particular construct or syntax.

The remainder of this paper is as follows. Section 2 presents the three methodological principles that motivate the goals. Section 3 constitutes the core scientific contribution: six named design goals, each defined by the problem it solves, the corresponding language decision, and its rationale. Section 4 establishes an overview of the tools currently developed for REQ. Section 5 concludes and outlines the research roadmap.

2 Methodological Foundations

The design goals that we will present in Section 3 are collectively motivated by three methodological principles aimed at mitigating the authoring gap described in Section 1.4. This section presents each principle, the engineering practice it is drawn from, and the design goals it motivates.

2.1 Requirements-as-Code

The requirements-as-code methodology is an adaptation of the infrastructure-as-code (IaC) paradigm to systems engineering, established in software DevOps. In IaC, infrastructure configurations are expressed as versioned source files, a practice exemplified by tools such as Terraform, such that the repository becomes the single authoritative state of the infrastructure, with no configuration existing outside it. Applied to systems engineering, this paradigm holds that the requirements specification should be treated as source code: plain text, versioned, and amenable to the same collaborative toolchain that governs software development.

This approach has not yet seen widespread adoption in industrial systems engineering, though the growing interest in textual modeling formats, evidenced by SysML v2 [13], signals an increasing openness to file-based specification workflows. The paradigm brings well-established benefits: continuous integration, fine-grained version control, and systematic peer review through pull requests. These practices have demonstrated their effectiveness for managing complex, multi-stakeholder artifacts in software projects and constitute a sound implementation of the *digital thread* concept. This principle motivates design goals G1, G3, and G5 presented in Section 3.

2.2 Progressive Formalization

A prominent disadvantage associated with model-based and formal verification practices is the upfront effort required to construct an adequate formal representation of the system [17, 16, 12]. Such representations are typically built from scratch, requiring significant expertise, and are derived manually from natural language requirements and existing implementation artifacts. All cross-artifact links must be established and maintained manually, with consistency re-validated at each development milestone.

The progressive formalization principle proposes that the level of formalism applied to a requirement should be a local, incremental decision rather than a global, upfront commitment. Engineers begin authoring at the level of natural language and introduce formal precision only where needed and when available, without this decision affecting the rest of the specification. The result is a spectrum of formal-

ism within a single document, ranging from informal prose to precisely typed expressions, coexisting without structural disruption. This approach, which we term *progressive formalization*, draws inspiration from the FORM-L language [18], and RSML [19] which established the paradigm of requirements as formal artifacts. The articulation as a first-class language design property is, to our knowledge, novel. This principle motivates design goals G2 and G4, presented in Section 3.

2.3 Early Automated Diagnostics

The third principle is methodological in origin: errors and inconsistencies in a specification are significantly less costly to resolve when they are detected at authoring time rather than at a later verification milestone [22, 9]. This motivates the design of the specification language so that automated diagnostic feedback is available as early and as broadly as possible across the spectrum of formalisms.

The achievable level of diagnostic coverage varies with the statement's level of formalism. Fully formal expressions admit precise, deterministic checking. Natural language statements, being undecidable, are amenable only to heuristic analysis. The methodological aspiration is that the language maximizes diagnostic coverage across this full spectrum, combining formal analysis where expressible and heuristic analysis where not, and that formal statements, where their formalism is sufficiently structured, be amenable to automated derivation into the adequate verification activity: model checking, constraint solving, or simulation. This principle motivates design goal G6 and acts as a cross-cutting constraint on the design of all other goals presented in Section 3.

3 Design Goals

This section presents the six design goals of the REQ language. Each goal is structured as follows: the engineering problem it addresses, how current approaches handle it and where they fall short, and the corresponding language design decision.

Where relevant, the goals will be illustrated by the example provided in Fig. 3. The example is minimal, focusing on the actual language property rather than its representational capabilities.

G1 – Requirements-centric

Requirements are the primary governance artifact of systems engineering projects [1, 20]. They serve as the basis for certification, contractual commitments, and stakeholder decision-making across the entire development lifecycle. Unlike modeling

```
package System
  part Wheel
    let deceleration_duration in real [s]
    let is_decelerating in boolean
    let speed in real [m/s]
  part

  requirement Wheel_deceleration is
    @@
      The [[Wheel]] shall decelerate within 3s.
    @@
  requirement

  requirement Wheel_deceleration_duration
    refines Wheel_deceleration is
    deceleration_duration = 3
  requirement

  requirement Stopped_after_deceleration
    refines Wheel_deceleration is
    is_decelerating implies eventually speed = 0
  requirement
package
```

Fig. 1 Example of a req based requirement specification for self-braking wheel

or formal artifacts, requirements are accessible to all stakeholders regardless of expertise in formal methods: the barrier to authoring a requirement is lower than that of a modeling notation or a formal specification language. This central role is not reflected in the tools currently available.

In ALM platforms such as DOORS [2] or Polarion [3], a requirement is a record in a database that tracks a plain-text field with metadata, which has no semantic role in the broader specification. In modeling languages such as SysML v2 [13], even though requirements exist as language concepts, requirements are often secondary elements imported from an ALM and allocated to model constructs; the model, not the requirement, is the primary unit of expression. Formal methods tools work with properties cast in a target verification formalism, not with requirements as first-class objects. In all three cases, the requirement is subordinate to the tool's primary artifact.

The design decision is to make the requirement the primary unit of expression in REQ. Every specification statement about the system or its environment is expressed as a requirement. Modeling constructs (G2) and formal expressions (G4) exist within requirements, not alongside them. A REQ specification is a structured collection of requirements, and a system implementing the specification must comply with each of the requirements. The specification acts as a contract in the sense of the design-by-contract theory [21]. This is what we mean by calling the language

requirements-centric. This goal is motivated by the requirements-as-code principle (Section 2.1).

G2 – Model-as-glossary

This goal is a direct consequence of G1: if requirements are the primary unit of expression, the model must emerge from within them rather than be maintained as a parallel artifact. Requirements in natural language constantly invoke system concepts: components, interfaces, physical quantities, and regulatory terms that remain undefined and unlinked in practice, constituting a major source of specification drift [10, 6].

In ALM platforms, glossaries are reduced to flat lists of acronyms, lacking semantic structure and any mechanism for referencing terms within requirement text. Terminology drifts as requirements evolve because the connection between a term and its occurrences is implicit. In modeling languages such as SysML v2 [13], entities are formally named and structured but maintained in a model decoupled from the requirements documents: changes to requirements do not propagate to the model, and model elements can be introduced or renamed without any traceability to the requirements that motivated them. Formal methods tools define the vocabulary of their models precisely, but only within the scope of the formalism, leaving the broader natural language requirements vocabulary unstructured.

The design decision is to replace the parallel model with a structured glossary embedded within the specification itself. Named concepts are declared as typed glossary elements and referenced explicitly within requirement text, making every term occurrence traceable to its definition. Each concept is defined in a single place, making drift detectable. The glossary is hierarchical: elements carry typed attributes and can be organized into taxonomies, forming a lightweight model of the system vocabulary that is, by construction, traceable to the requirements that reference it. When deeper structural analysis is required, this glossary model can be exported to a modeling language. This goal is motivated by the progressive formalization principle (Section 2.2). An example of a glossary-based modeling pattern is shown in Fig. 3 for the natural language requirement `Wheel_deceleration`.

Specification and modeling languages differ significantly in the richness of their construct vocabularies. A richer vocabulary enables more precise structural representations but imposes a learning investment and, more fundamentally, commits the author to a particular engineering paradigm before the requirements have been fully understood.

Modeling languages such as SysML v2 [13] provide extensive sets of constructs: parts, ports, flows, actions, state machines, parametric constraints, and allocation tables, each carrying specific engineering semantics. While this richness enables detailed structural and behavioral modeling, it requires authors to map their system understanding onto those constructs, which may not align with the domain's conceptual vocabulary. The learning curve is steep, and requirements that do not fit

naturally into the available constructs must be reformulated, risking loss of meaning. Alternatively, a language variant can be developed, but this would defeat the purpose of a unified language [15]. ALM platforms avoid this problem by providing no constructs at all, but then lose the ability to impose any shared structure on the specification. Formal methods tools define their own mathematical objects (automata, process terms, transition systems), which are even narrower in scope.

The design decision is for REQ to provide the smallest construct set sufficient to express a requirements specification: `package` for specification structure, `part` and `attribute` for glossary and ontology elements, and `requirement` as the primary unit (G1). No architectural, behavioral, or domain-specific constructs are added. This minimal vocabulary is accessible to any engineer who can write requirements, imposes no paradigm commitment, and is applicable to any engineering domain without adaptation. This goal is motivated by the requirements-as-code principle (Section 2.1).

G4 – Formalism plurality

Where G3 concerns the structural constructs the language provides, G4 concerns the formal content that can appear within those constructs. The requirements of a cyber-physical system span heterogeneous engineering disciplines, each with its own verification methods and mathematical vocabulary. A timing constraint is naturally expressed in temporal logic and verified by a model checker; a physical behavior is naturally expressed as a differential equation and verified by simulation; a logical constraint is naturally expressed in propositional logic and verified by constraint propagation. No single formalism covers the full scope [11].

Modeling languages commit to a specific behavioral formalism: SysML v2 [13] expresses behavior through state machines and action flows, which favor executability but cannot faithfully represent requirements whose nature is incompatible with those constructs. Formal methods tools each provide exhaustive guarantees within their respective formalisms, but at the cost of restricting the scope of verifiable requirements to what fits their mathematical models. In both cases, the specification must be approximated to fit the tool, introducing simplifications that may not meet the required level of verification precision. ALM platforms impose no formalism at all, providing neither the expressiveness of formal notation nor the verifiability that formal notation enables.

The design decision is that REQ imposes no constraint on the formalism used within individual requirements. A typical progressive formalization workflow begins with plain natural language requirements, which are then refined or derived into formal requirements as the system is better understood and verification targets are identified. The formalism applied to each requirement is chosen according to the nature of the system element it concerns and the property to be verified. The language is designed so that formalisms are syntactically recognizable: temporal operators signal eligibility for model checking, real-valued equations signal eligi-

bility for simulation or numerical analysis, propositional expressions signal eligibility for constraint solving. This syntactic recognition enables automated identification of the applicable verification method for each formal requirement. For each requirement, including plain-text ones, heuristic static analysis is applied as a universal baseline. The separation between this heuristic baseline and the deterministic, formalism-specific routing is a key property of this goal. This goal is motivated by the progressive formalization principle (Section 2.2) and the early automated verification principle (Section 2.3). An example of formalism plurality is shown in Fig. 3; the requirements presented are of several natures: natural language, algebraic, and temporal logic-based.

G5 – Modularity and standardization

Systems engineers rarely write specifications from scratch. System families share generic requirements, organizations maintain authoring standards and domain ontologies, and certification frameworks impose mandatory requirement patterns. A specification language must support structured reuse of these artifacts across projects and organizations, with a level of semantic precision that makes the reused content meaningful rather than merely textual [16].

ALM platforms provide template mechanisms for plain-text requirement patterns, which standardize authoring style but carry no formal semantics: a template is a textual skeleton with no machine-verifiable structure, and instantiating it correctly is left entirely to the author. NASA FRET [8] provides structured requirement patterns with formal semantics but constrains authors to its own formalism and cannot accommodate the plurality of formalisms of G4. SysML v2 [13] packages provide structural reuse, but the reused content is bound to SysML v2's construct vocabulary: a consumer must adopt the same paradigm to make use of a shared package, and the natural language content of requirements remains semantically unstructured in either case. None of these approaches supports cross-organization, toolchain-agnostic library sharing with mixed levels of formalism.

The design decision is to provide a parametric template mechanism applicable to both requirement patterns and glossary elements, analogous to generics in programming languages. A template may be instantiated with plain text, formally typed content, or mixed content, preserving the flexibility of G3 and G4 while enabling a reusable specification library. Domain ontologies are constructed by composing typed glossary templates; organizational or regulatory writing standards are encoded as requirement templates. Libraries are distributed as plain-text source files, requiring no specific toolchain for consumption or distribution. This goal is motivated by the requirements-as-code principle (Section 2.1).

G6 – Formal traceability

Traceability between requirements is a pervasive obligation in systems engineering, arising in multiple distinct contexts. In client/supplier relationships, every item in a request for quotation or a system-level requirement set must be traceable to a specification element that addresses it [1], forming the contractual basis for compliance demonstration. System validation activities require that every stakeholder requirement be traceable to evidence of its satisfaction [1]. In safety-critical domains, certification frameworks additionally mandate specific traceability patterns as part of the design assurance process [4, 5]. In all cases, traceability links form a network that enables stakeholders to assess coverage, evaluate the impact of changes, and justify engineering decisions to internal and external reviewers.

In ALM platforms, traceability is implemented as labeled links between plain-text records. Tools enforce referential integrity (a link target must exist) but not semantic consistency. Three classes of problem consequently go undetected: *missing links*, where a requirement has no downstream coverage and is therefore not demonstrably addressed; *obsolete links*, where a linked requirement has since been modified or deleted, rendering the link misleading; and *inconsistent links*, where the two linked requirements have no discernible semantic relationship. NLP- and LLM-based approaches can surface candidate links or flag suspicious ones [7], but provide no deterministic guarantee and cannot substitute for formal validation. In modeling languages, traceability between model elements is correct by construction within the model, but it remains disconnected from the natural-language requirements from which the model was derived, leaving the requirements-to-model link unverified.

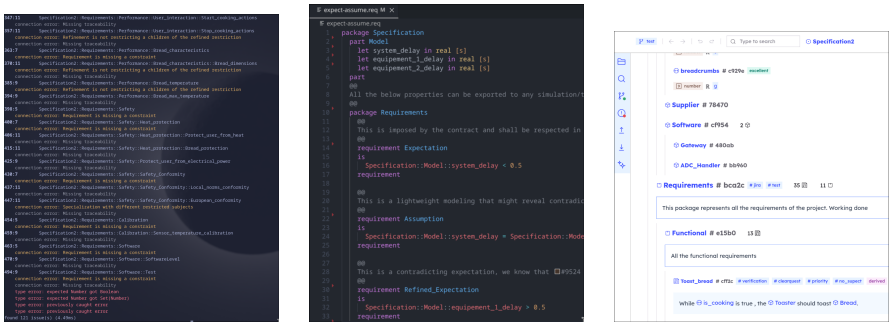
The design decision is to ground traceability in the shared glossary introduced by G2. A refinement link between two requirements is checked for consistency against the typed glossary elements they reference: a requirement can only refine another if the glossary terms it introduces are specializations of those present in the parent. This provides a deterministic, compiler-verifiable consistency criterion that does not exist in plain-text systems, while remaining grounded in the requirements themselves rather than requiring a parallel formal model. This goal is motivated by the early automated diagnostics principle (Section 2.3). Traceability links are shown in Fig. 3, where the `refines` link have a well-defined semantic helping to ensure the consistency of the link.

4 Available tools

The design goals of Section 3 are supported by three tools, illustrated in Fig. 2; full documentation is available at `https://req-lang.org`.

`reqtool` is the reference compiler: it performs syntax validation, reference resolution, and heuristic diagnostics covering missing references, traceability inconsistencies, and style violations. A VS Code extension provides syntax highlighting

and will bundle `reqtool` for inline diagnostic reporting. Qitab is the reference graphical editor; it offers structured visualization and editing of packages, glossary elements, and traceability links, with import/export to standard artifact formats for round-trip with existing ALM workflows (`https://qitab.io`).



(a) `reqtool` compiler output (b) VS Code extension (c) Qitab graphical editor

Fig. 2 The REQ toolchain: (a) static analysis diagnostics produced by `reqtool`, (b) inline authoring feedback via the VS Code extension, and (c) structured specification visualization in Qitab.

5 Conclusion

This paper has presented six named design goals for REQ, a requirements specification language designed to close the authoring gap between informal engineering documents and formal V&V back-ends. The goals — requirements-centricity (G1), model-as-glossary (G2), construct minimality (G3), formalism plurality (G4), modularity and standardization (G5), and formal traceability (G6) — form a coherent design rationale grounded in the three methodological principles of Section 2: requirements-as-code, progressive formalization, and early automated diagnostics.

The current toolchain, described in Section 4, covers static analysis and reference resolution. Formal proof, model checking, and simulation are not yet supported; these verification activities are on the research roadmap, and the language is explicitly designed to feed dedicated back-ends for each.

Further work will address three verification pillars: static analysis extensions, behavioral contract integration for co-simulation, and connections to model-checking back-ends.

References

1. ISO/IEC/IEEE: Systems and software engineering — System life cycle processes. Standard 15288:2018. International Organization for Standardization
2. IBM Corporation: IBM Engineering Requirements Management DOORS.
3. Siemens Digital Industries Software: Polarion ALM.
4. SAE International: Guidelines for development of civil aircraft and systems. SAE ARP4754A. SAE International, Warrendale (2010)
5. ISO: Road vehicles — Functional safety. Standard ISO 26262:2018. International Organization for Standardization
6. Méndez Fernández & Wagner (2014) — Naming the Pain in Requirements Engineering
7. Liping Zhao, Waad Alhoshan, Alessio Ferrari, Keletso J. Letsholo, Muideen A. Ajagbe, Erol-Valeriu Chioasca, and Riza T. Batista-Navarro. 2021. Natural Language Processing for Requirements Engineering: A Systematic Mapping Study. ACM Comput. Surv. 54, 3, Article 55 (April 2022), 41 pages. https://doi.org/10.1145/3444689
8. Giannakopoulou D, Pressburger T, Mavridou A, Rhein J, Schumann J, Shi N (2020) Formal requirements elicitation with FRET. In: REFSQ 2020. Springer, Cham
9. Elizabeth Hull, Ken Jackson & Jeremy Dick, Requirements Engineering, 2004
10. Tukur, Muhammad & Umar, Sani & Hassine, Jameleddine. (2021). Requirement Engineering Challenges: A Systematic Mapping Study on the Academic and the Industrial Perspective. Arabian Journal for Science and Engineering. 46. 3723-3748. 10.1007/s13369-020-05159-1.
11. Jean-Michel Bruel, Sophie Ebersold, Florian Galinier, Manuel Mazzara, Alexandr Naumchev, and Bertrand Meyer. 2019. The role of formalism in system requirements (full version). 1, 1, Article 1 (November 2019)
12. Chami, M., & Bruel, J. (2018). A Survey on MBSE Adoption Challenges.
13. Object Management Group: Systems Modeling Language (SysML) v2.0. OMG Document ptc/2024-04-01 (2024)
14. Object Management Group: Kernel Modeling Language (KerML) v1.0. OMG Document ptc/2024-04-01 (2024)
15. Jansen, Nico & Pfeiffe, Jérôme & Rumpe, Bernhard & Schmalzing, David & Wortmann, Andreas. (2022). The Language of SysML v2 under the Magnifying Glass.. The Journal of Object Technology. 21. 3:1. 10.5381/jot.2022.21.3.a11.
16. Henderson K, McDermott T, Salado A. MBSE adoption experiences in organizations: Lessons learned. Systems Engineering. 2024;27:214–239.
17. Hubert Garavel, Maurice H. ter Beek, and Jaco van de Pol. 2020. The 2020 Expert Survey on Formal Methods. In Formal Methods for Industrial Critical Systems: 25th International Conference, FMICS 2020, Vienna, Austria, September 2–3, 2020, Proceedings. Springer-Verlag, Berlin, Heidelberg, 3–69.
18. Nguyen, Thuy. (2019). Formal Requirements and Constraints Modelling in FORM-L for the Engineering of Complex Socio-Technical Systems. 123-132. 10.1109/REW.2019.00027.
19. Florian Galinier, Sophie Ebersold, Jean-Michel Bruel. Requirements Specific Modeling Language : un langage formel d'expression d'exigences. Conférence en Ingénierie du Logiciel, Jun 2018, Grenoble, France. ⟨hal- 01815467⟩
20. INCOSE Systems Engineering Handbook, 5th Edition, ISBN: 978-1-119-81429-0
21. Albert Benveniste, Benoît Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, et al.. Contracts for Systems Design: Theory. [Research Report] RR-8759, Inria Rennes Bretagne Atlantique; INRIA. 2015, pp.86. ffhal-01178467f
22. Boehm B, Basili VR (2001) Software Defect Reduction Top 10 List. IEEE Computer 34(1).