Review and Challenges of Assumptions in Software Development

Md Abdullah Al Mamun and Jörgen Hansson
Software Engineering Division
Department of Computer Science & Engineering
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden
{abdullah.mamun, jorgen.hansson}@chalmers.se

Abstract—**The problems of implicit and invalid assumptions have been identified as one of the key reasons to project and software failures. Assumptions are available in almost all aspects of the software development from human factors to different software development activities. They also have influence on software quality attributes. The aim of this article is to provide a review of existing work in assumption management and find out the assumptions related challenges that should be mitigated in order to build better systems. The results show that assumptions are concerned with many different areas of software engineering and that existing approaches suffer from the lack of scope of assumptions categories and some concerns that are impacted by the assumptions. We believe a holistic assumption management approach can mitigate assumptions related challenges by integrating concerned areas and contribute to build systems with smooth software integration and evolution.**

Keywords- assumptions; assumption management; software evolution; software and system integration; cyber-physical system

I. INTRODUCTION

Today's Cyber-Physical Systems (CPSs) intrinsically combine many domains and areas of expertise in order to achieve system goals and maximize the benefit. This demands significant interactions among people, environment, software, and hardware artifacts, which in turn dramatically increases the complexity of the system. The maximum number of concerns that the human brain can consciously process at the same time is limited. It is thus challenging for the software and system developers to consider all significant assumptions and constraints among various components to make good decisions.

An assumption is a statement that is assumed to be true and it is invalid when the assumed statement is actually not true. Assumptions are implicit when they are not documented. Assumptions can be implicit in at least two ways. First, when people are aware of the assumptions but do not document them because of lack of consciousness about the pitfalls of implicit assumptions or due to political reasons within the organization. Second, there is no awareness of the (implicit) assumptions among the stakeholders/people. Implicit assumptions thus represent tacit knowledge, which has been identified by the knowledge engineering discipline [10] as volatile and challenging to preserve and transfer.

When implicit assumptions are not documented, they get lost over time. This might happen due to that the architects forget about the assumptions they made in the past, or that the architects are not available at present. The gradual loss of architectural knowledge is a problem in the area of software architecture and this problem is known as architectural erosion or architectural drift. This scenario is also applicable for requirements, coding, and testing.

Invalid assumptions are reported as the root cause of system and project failure [17, 40, 8]. In practice, people do not make invalid assumptions intentionally or because of lack of knowledge. Today's systems often work in a complex dynamic environment with the presence of reusable components. Along with many benefits, reusable components also bring certain challenges. As COTS are developed to work in different environments, they generally do not offer a perfect match with a specific use. The use of a system as a part of a larger system is common as it is not feasible to build everything from scratch. However, a system is not always built with the intention of it being used as a part of a larger system. Moreover reusable components, COTS, middleware might have their own sets of assumptions that are implicit or not visible to the people using them. Hence, people can make assumptions about different components that are conflicting, or mismatched, with existing assumptions. The development of complex systems deals with various domains where every domain has its own practices. However, in reality an organization cannot always adopt necessary software practices of the particular domains they are engaged with. Thus, an organization might apply their existing software practices onto a new domain.

The review shows that there is a broad landscape of assumptions and several challenges have been recognized in certain areas. Initial work has been conducted to address some challenges, but we observe that there is a lack of integrated approaches toward systematic assumption management. Successful mitigation of these challenges would indeed support virtual integration of components, continuous deployment, and more loosely coupled CPSs development.

The organization of this report is as follows. Section II defines assumptions and some assumptions types used in this article, and explains how requirements, constraints, assumptions, and design rationales are connected to each other in an interchangeable way. Section III shows the literature

review of assumptions in software engineering. The challenges of assumptions are presented in section IV followed by the summary.

II. BACKGROUND

A. Definition of Assumption

The WordNet[1] lexical database defines the term assumption as follows:

- "A statement that is assumed to be true and from which a conclusion can be drawn". Example: "on the assumption that he has been injured we can infer that he will not play".
- "A hypothesis that is taken for granted". Example: "any society is built upon certain assumptions".
- "The act of assuming or taking for granted". Example: "your assumption that I would agree was unwarranted".

Now we define some assumptions types used in this paper.

Invalid vs. Valid assumptions: An assumption is considered invalid if a stated assumption is false or incorrect; it is valid otherwise, i.e., the stated assumption holds. The validity/invalidity of an assumption can most often be determined by verifying its fact without necessarily looking at any other assumptions. On the other hand, *conflicting* and *mismatched* assumptions are determined from the conjugation of more than one assumption.

Conflicting assumptions: An assumption is conflicting, if it contradicts or conflicts with one or more other assumptions. It can be both invalid and valid.

Mismatched assumptions: An assumption *X* is mismatched, if we cannot determine whether the associated components/artifacts of *X* would fulfill the fact of *X*. In other words, there is no evidence provided by the components/artifacts that could be matched against what is assumed. It can be both invalid and valid.

B. Requirements, Constraints, Assumptions, Design Rationale

It is difficult to distinctly divide requirements, constraints, assumptions, and design rationales as they often overlap each other. Sometimes these terms are used interchangeably [27] and sometimes broadly [19] to extend the coverage of their definition. In general, requirements are the expectations of the customers about a system. Constraints are facts that impose restrictions, limitations, regulations on a system. In a classical sense, requirements and constraints are sets that both the software development organization and the customers agree upon. Requirements and constraints can be both functional and nonfunctional and it is a common organizational practice to document them.

Assumptions are statements that are assumed to be true. Assumptions build the underlying reasons behind the decisions where a decision can be an architectural decision etc [35]. When decisions along with their underlying assumptions are implemented, assumptions act like constraints by restricting, limiting, and regulating the system. From this point of view, assumptions and constraints are similar. However, assumptions can be invalid from the very beginning of their existence, which is not applicable for the constraints. On the contrary, both assumptions and constraints can be invalid at any time in the future when the system evolves.

Design rationales are the motivations of design decisions where a collection of design decision explains why an architecture is in a certain form. Kruchten et al. [35] do not distinguish between assumptions and design rationale since it is difficult to make a clear distinction and consider assumptions as general denominator for the forces driving architectural design decisions. Both assumptions and rationales can be considered as elements of design decision [43]. In contrast, assumptions and constraints can also be considered as elements to capture design rationales [21].

III. ASSUMPTIONS IN SOFTWARE ENGINEERING

Figure 1 shows how assumptions can be scattered in different software development phases and still connected. A single assumption can be associated with artifacts at different levels. The scenario becomes much more complicated when assumptions are connected with requirements, design decisions, design rationales and other possible knowledge categories that are influenced by the assumptions. This review also finds assumptions in different areas of software engineering. Among them, software architecture is the area where assumptions are mostly used. Some work is directed toward implementations. Work has also been conducted in the security domain, especially at the requirements engineering level, as well as in the architectural knowledge management areas; here most often assumptions are treated informally in the form of free text.

This section first presents different *assumptions modeling approaches* then *architectural mismatch* problems due to the assumptions, followed by assumptions in the area of *requirements engineering* and *software security*. The end of this section focuses on assumptions in the *knowledge management* discipline and *software development processes*.

A. Assumption Modeling

There have been attempts on modeling assumptions. We have divided them into two classes that are formal and semi-formal. By formal, we mean those modeling approaches that formalize the statement or the fact of an assumption along with other attributes. Approaches that describe the fact/statement of the assumptions as free text but other attributes like assumption category description, source, impact, criticality, tractability information etc are structured are termed as semi-formal.

[1] Website: http://wordnetweb.princeton.edu/perl/webwn

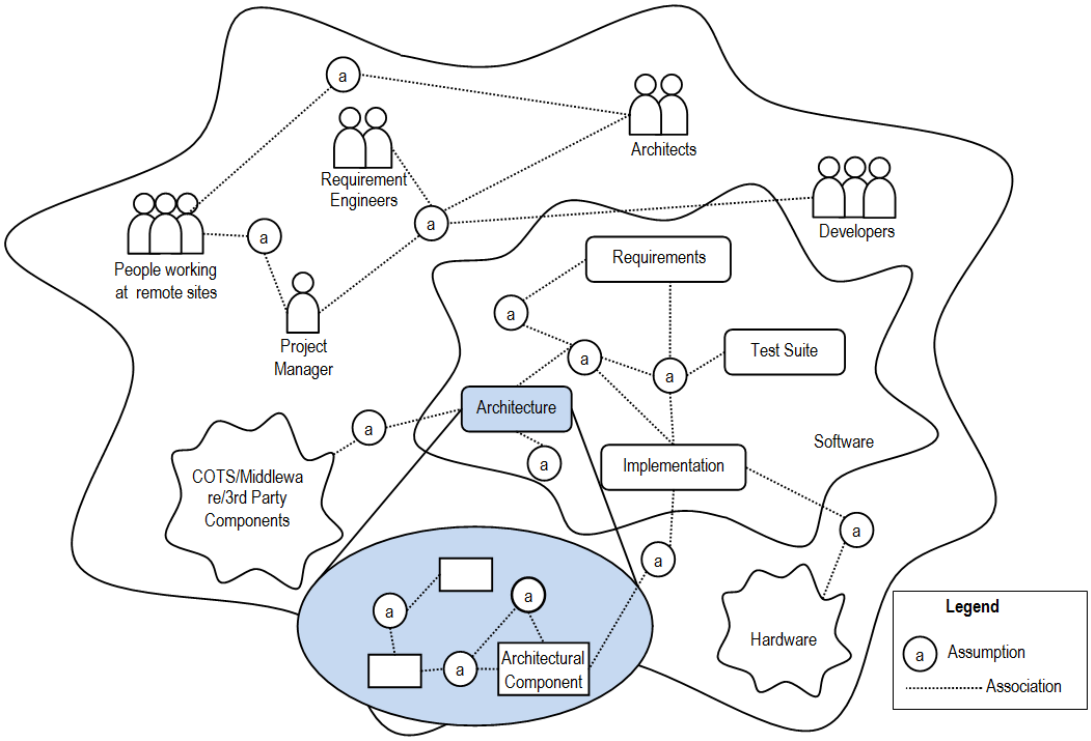

Figure 1. Assumptions in Software and System Development

The idea of informal assumptions modeling is not so appropriate. However, we can say assumptions are informal when they are documented without proper structure, completely in free text in the form of comments. The advantage of formally modeled assumptions is that they are potentially machine-checkable. However, formal approaches have suffered from limited scope, as it is challenging to formally model assumptions of concerning, e.g., managerial and environmental aspects.

1) Formal Approaches

A framework toward assumption management has been developed by Tirumala [2]. They have developed a language for documenting assumptions in a machine-checkable format where the assumptions and the guarantees for the assumptions are encoded as part of the architectural components. The framework is capable of dealing with both the architecture and the implementation. In addition to the static assumptions, the framework also supports dynamic validation of assumptions. Even though, automated checking of invalid assumptions is a big advantage for the large-scale, complex system development but the scope of the targeted assumptions is limited to the technical category. The framework is implemented for the Architecture Analysis and Design Language (AADL) [31], which is an Architecture Description Language (ADL)

2) Semi-Formal Approaches

Lewis et al. [14] have developed a simple assumption management system prototype for recording and extracting assumptions from code written in Java into a repository. The assumptions are written in the code using XML and saved into the repository using an assumption extractor. This web-based assumption management system offers browsing and searching of assumptions with given criteria. The stored assumptions are then reviewed by a person who acts as a validator. The management system also maintains system and project related information like users, roles, projects and types of assumptions. The scope of this prototype is limited to assumption management at the implementation level.

A meta-model for explicating assumptions in the software architecture has been developed by Lago and Vliet [33]. The model is able to handle assumption dependencies between the product feature model and the architectural model. They have worked with a software product family architecture implementing variability to achieve flexibility. They introduce the term invariability and argue that invariability should also be modeled along with variability to let the model express what cannot be changed. As assumptions are somewhat related to the constraints that impose limitations on the system behavior, assumptions would tell us what we cannot change or what would be challenging to change. Thus it is possible to achieve architectural invariability through explicit assumptions modeling.

Ordibehesht [16] has implemented an assumptions modeling method for explicating assumptions in the AADL. The modeling method consists of an assumption specification meta-model for structuring assumptions information and an assumptions specification approach to specify the meta-model together with the architecture descriptions. The meta-model contains dependency information between the assumptions and the architectural components in order to facilitate traceability.

B. Architectural Mismatch

Garlan et al. [8] report that implicit assumptions are the root cause of widespread software reuse problems. They used the term "architectural mismatch" to express the idea that reusable architectural components make implicit assumptions on other parts of the architecture without being validated. As these assumptions are implicit, they often conflict others thus making the system unusable. Since the assumptions are usually implicit, it is difficult to analyze them before the system is built. They identify four categories of assumptions that can contribute to architectural mismatch in terms of components and connectors. They are:

- Nature of the components (control model, data model)
- Nature of the connectors (protocol, data model)
- Global architecture structure
- Construction process (development environment and built)

Even though the authors identify implicit assumptions as the root cause of the architectural mismatch and suggest to make assumptions explicit, they believe explicating assumptions alone cannot completely fight architectural mismatch, and thus they suggest additional solutions such as use of orthogonal sub-components, create bridging techniques like mediators, and develop source of design guidance.

Garlan et al. [9] have revisited the challenges of architectural mismatch with the advancement of time in comparison to their previous study. They discuss three basic techniques with architectural mismatch namely mismatch prevention, mismatch detection and mismatch repairing along with the approaches to solve mismatch problems. Trust, dynamism, architecture evolution, and architecture lock-in are reported as new challenges in this field.

Cai et al. [25] have proposed an approach to identify architectural mismatches resulting from invalid assumptions in an event-based system. The semi-automatic approach is implemented with a customized version of the model checker Bandera/Bogor tool pipeline applied on a Java version of SIENA event service.

Architectural mismatch has been analyzed considering assumptions from a different perspective by Uchitel and Yankelevich [42]. Assumptions are viewed as connections that components make about each other. Here the term connection means something beyond the classical sense of connections. This view originated from Parnas [11] who extended the view of the connections from control or information transfer points to components' associations. Uchitel and Yankelevich [42] have performed behavior analysis to detect architectural mismatches where a labeled transition system (LTS) is used to model process behavior. They also discuss some issues of extending ADLs to include assumptions.

Lemos et al. [36] have focused on architectural mismatch tolerance, i.e., an approach that would help the system to tolerate architectural mismatches during run-time rather than dealing with the mismatches during the development time. They apply the general principle of fault tolerance to deal with architectural mismatches.

C. Assumptions and Requirements

It is generally considered that a large number of assumptions lie around the requirements to weave a complete picture of the system. Lamsweerde [4] describes this scenario as 'assumptions underlie the requirements iceberg'. Thus assumption management is often considered to be closely intertwined with the field of requirements engineering.

Fickas and Feather [41] argue that invalid environmental assumptions cause the system to evolve. Therefore, it is interesting to know when such assumptions get invalid while the system is executing. The concept of this approach is to monitor the underlying assumptions of the requirements to realize whether the requirements are met by the system while it is executing. When certain assumptions are found invalid, their corresponding/dependant requirements are considered to be compromised.

A temporal mathematical model has been proposed by Miranskyy et al. [1] to describe the relationship between requirements and assumptions in the context of risk prediction associated with assumptions failure. In the proposed model, the relationship between requirements and assumptions are captured using a Boolean network. A stochastic process is used to model the validity of the system over time.

D. Software Security

Assumptions in the security domain are known as trust assumptions because trust and trustworthiness build the foundations of security [20]. Trust assumptions might have significant impact on the system's security. An example of an implicit or explicit trust assumption can be *compilers are not vulnerable to systems security*. However, this assumption can be invalid as Thompson [23] shows how the compilers can introduce trapdoors to compromise the security of a system. Thus it needs to be reviewed for its validity. Viega et al. [22] discuss the potential risks of trust assumptions towards the software security and the origins of invalid trust assumptions like user input, client application, execution environment, software developers, users etc. They suggest adopting a general assumption management strategy, i.e., assumption identification, documentation, and analysis in order to minimize the security risks due to invalid assumptions.

Haley et al. [5] discuss trust assumptions from the view of the requirements engineers in the context of security. The requirements engineers make assumptions when analyzing the security requirements. The scope of the analysis, security requirements' derivation, and in some cases how functionality is realized are affected by the trust assumptions. A representation of trust assumptions is showed by Haley et al. [5] along with a case study to examine the impact of trust assumptions on software using secure electronic transaction specification. In more recent work, Haley et al. [6] attempted to answer to the question "how to determine adequate security requirements for a system". In this work, they considered assumptions as one of the three criteria that should be satisfied to determine adequate security requirements. A lightweight approach for mitigating security risks based on trust assumptions is proposed by Page et al. [44] that can be used within agile development environments. This work is directed

toward detection and mitigation of security risks. They developed a model where the concept of trust assumptions is used to derive obstacles, and the concept of misuse cases is used to model the obstacles.

E. Architectural Design Decision & Rationale Management

The literature of architectural knowledge identifies four primary views on architectural knowledge namely pattern, dynamics, requirements and decision-centric view [37]. The decision-centric view emerged as the importance of preserving architectural design decisions and rationales behind the decisions were realized [24, 20], and there seems to be a gradual shift of viewing software architecture as the high-level structure of components and connectors (i.e. the end result) to the rationale behind the end result [37].

Since an architecture is built based on certain design decisions, it is also seen as a collection of design decisions. Going a level further down, every design decision is made based on some rationales. Thus design decisions along with the rationales explain why an architecture is in a certain form. Assumptions are considered in both design decision and design rationale management. With the change of the perspective, something identified as an assumption may be seen as a design decision. It should be noted that, e.g., Kruchten et al. [35] do not distinguish between assumptions and design rationale as they have found it difficult to make a clear distinction. Rather, assumptions are seen as general denominator for the forces driving architectural design decisions [35].

Both assumptions and rationales are considered as elements of design decision by Dingsøyr and Vliet [44]. They describe assumption as the underlying facts about the environment in which the design decision is taken and rationale as the explanation of why the specific decision was taken. A pragmatic approach to capture design rationale has been proposed by Tyree and Akerman [21]. They include assumptions and constraints along with other elements in a template developed to capture design rationale. A rationale-based architecture model has been developed by Tang [3], which represents design rationale, design objects and their relationships. The model is able to capture both qualitative and quantitative design rationale where assumptions and constraints are considered as the drivers of design rationale. Even though assumptions are reported as one of the key factors driving design decisions and rationales, the discussed literature captures assumptions as text in natural language without further structure.

F. Other Work

Ostacchini and Wermelinger [18] have experimented with a lightweight assumption management method on agile development over three months with the result that assumption management can be integrated with the agile developments. They have recorded 50 assumptions where more than 50% are organizational or managerial. They suggest engaging the management/managers into the assumption management process as over 15% of the managerial assumptions were identified as invalid during the three months observation period.

Roeller et al. [38] have worked on the recovery of assumptions from a system that was built in the past without the assumptions being documented. At first, they reviewed financial reports, documentations, development process information extracted from the version control system and source code to identify error prone modules in the studied system. Tools were used to extract different metrics from the version control data and source code. Furthermore, interviews were performed with the architects and the developers to discuss the selected modules in order to capture the implicit assumptions.

The authors express that it is challenging to recover assumptions from a system without having a thorough understanding of the system. Moreover, the unavailability of key people and stakeholders, change in responsibilities in the project, and identifying the right person knowledgeable to specific artifacts are also quite challenging to deal with. Similarly, Garlan et al. [8] have reported from their experience with AESOP that assumption recovery could be expensive thus impractical or even impossible for legacy systems when the source code is not available.

IV. Challenges of Assumptions

CPSs are multidisciplinary in nature, addressing engineering issues at software, system, and mechanical level. Furthermore, CPSs demand a lot of interaction among the concerned components and environments. They are often also highly complex and tightly coupled systems. The development of CPSs is also often distributed in nature. Thus, it is unrealistic or not feasible to co-locate the entire CPSs development process. It is thus desirable to facilitate stronger integration approaches, and weaken strong coupling and dependencies in the development as well as the architecture level. Powerful management of assumptions has a strong potential in addressing identified concerns, e.g. the concepts of *assumptions-aware components* and *separating assumptions from artifacts (section A and B)*. Other challenges identified include evidence-*based software engineering* (section C) and *assumptions in the organization's safety culture* (section D). The *holistic assumption management system* (section E) is the foundation to manage assumptions in an efficient way throughout the entire software development process. *Assumption-based trust building* (section H) concept is applicable for human factor *trust* in global software developments thus it would also support the development of CPSs.

A. Assumption-Aware Component Development

CPSs are characteristically tightly coupled, which incurs certain inflexibilities in the system development. It would be desirable to develop components more independently, e.g., loosely coupled but still composable, which could be facilitated through self-descriptive architectural components or executable software components. The assumption-aware components should be able to describe their own structure and details about what the components expects from the other components and what the components provide for the other components with a standard syntax that is understandable by all the parties. As assumptions build the leaf-level knowledge of the artifacts, it is

possible to encode the inter-component dependencies and relationships as assumptions into the components. This makes the components assumption-aware and would offer better static and dynamic composability of such components by minimizing architectural or component mismatches. Assumption-aware components would also support the concept of virtual integration [32]. When the architecture and its substructures (e.g., components) are designed as assumption-cognizant, continuous deployment would be possible with the presence of monitors that would look for assumption-based conflicts and mismatches among the components. During composition, it is obvious that architectural mismatches may occur due to conflicting or mismatched assumptions that can be mitigated to some extent with the architectural mismatch tolerance techniques.

B. Separation of Assumptions from Artifacts

With the advent of assumption-aware component development, COTS and middleware developers would prefer to supply the assumptions related to the COTS or middleware to the customers without supplying the actual architecture or code so that the COTS or middleware can be tested whether they are composable with the customers' system or not. This concept also supports virtual integration.

C. Evidence-Based Software Engineering

In software engineering, *complacency* is a challenging problem to tackle. People suffer from complacency because of the lack of evidence. A study of five major spacecraft accidents reports complacency as the root cause of the studied system failures [29]. Another extensive study reports "*lack of evidence*" as the key problems of dependable software systems [12]. This study also proposes the idea of certifiably dependable systems that means a dependable system can be certified according to the available evidences supporting the dependability claims.

Assumptions build the leaf-level knowledge and forensic evidences of any artifact. They are able to reason why an architectural component is in a certain form. They can explain why a variable is not memory protected. In fact, assumptions make the leaf-level fingerprints of the decisions that we make while developing a system. From this point of view, assumptions are underlying evidences that can be used as a metric to measure the dependability of software components or systems.

D. Assumptions in the Organization's Safety Culture

The space shuttle Challenger disaster is a well-known case of system failure due to mismatched assumptions. The investigation report [40] of this disaster shows why assumptions should be added to the organization's safety culture. Before the launch of the shuttle, the engineers warned about the mismatched assumptions, which the management repeatedly ignored. The flight was already delayed with different issues. The management was afraid to delay it further probably because project delay negatively shows the efficiency of the management. Again, twelve years after the Challenger disaster, in 1996, we observed the explosion of Ariane 5 during its maiden flight. Such cases suggest considering assumptions in the organization's safety culture. An invalid/conflicting/mismatched assumption should be considered in the decision support system according to its criticality, priority, and impact. Assumption-based hazard analysis and Failure Mode Effect Analysis (FMEA) can reduce the risks of invalid, conflicting and mismatched assumptions.

E. A Holistic Assumption Management Approach

The span of assumptions in software development is not limited to any specific phases rather it is widespread. Available assumption management frameworks either focus on a narrow scope of assumptions types being very formal by modeling the assumptions in a machine-checkable format [2] or cover a wide variety of assumptions in a semi-formal approach that is not automated [33]. Moreover, existing approaches are not capable of providing an integrated solution toward assumptions by covering different software development phases, domains, COTS, and middleware. Since it is challenging and even not feasible to formally model all types of assumptions, we believe that building a flexible assumption management framework that would facilitate documenting assumptions in both machine-checkable and human-readable format is necessary.

In addition to the assumption-based services provided to different software development phases, the assumption management framework should also provide services to other frameworks that might be benefited from the assumptions, e.g., knowledge management, security, safety, etc. The knowledge management frameworks use assumptions as the underlying motivators for the design decisions or design rationales. For example, the assumption management framework can send a warning message to a knowledge management framework indicating that some of the design decisions or design rationales are subject to review, because their underlying assumptions are identified as invalid or conflicting.

F. Prioritization of Assumptions

A commonly raised argument against documenting assumption probably is "*assumptions can be anywhere; it is not feasible to document and maintain all of them simply because they are too many*". This argument is realistic and probably true. However, given the history we know that invalid assumptions can result in catastrophic consequences. While it seems infeasible to document all assumptions given time and budgetary constraints, we need to develop methods to prioritize assumptions in order to maximize the benefit over the cost.

Prioritization methods can help to identify/document important assumptions. They can also help selecting important assumptions among the identified assumptions that would be maintained throughout the software life cycle. Assumptions can be prioritized according to the domains, project types, technology used to build software system, software process, criticality, etc.

G. Assumption-based Verification and Validation

Assumptions are reported as one of the key problems failing the systems. It is expected that managing assumptions would reveal many defects earlier. Assumptions that are formally documented in the source code can be automatically

checked both statically and dynamically. However, there is no guarantee that a manual checking of assumptions in the implementation or automated/manual checking of architectural assumptions would prevent all defects related to these checked assumptions. From the testers' point of view, a good place to sniff the system for possible defects is where the defects are. However, it is not easy to know beforehand where the defects actually are. When finding the defects, the earlier in the system development life cycle is generally better. Therefore, it is quite reasonable to develop test cases motivated by the assumptions and then test the system with them. If we can automate these tests, they can be applied repeatedly in a cost effective way as the system evolves. Moreover, the architectural assumptions can be used to develop test cases, scenarios to review the architecture. The same concept is applicable for the verification of requirements.

H. Assumption-Based Trust Building and Maintenance

Software development is human-centric which involves a dimension of complexities toward successful management of software projects. The increasing popularity of distributed software development further boosts these complexities. In a globally distributed project people from different geographical location, society, culture, organizations and time zones take part in developing software.

Trust has been identified as one of the key success factors of distributed software projects [26, 28]. Face-to-face meeting and socialization are primary trust building activities that are easily achievable for the co-located team members. Time and budgetary constraints often do not allow face-to-face meeting among the distributed teams [30]. In distributed software projects, temporal, geographical and socio-cultural disparities obstruct communication, coordination and cooperation among the remote team members [34], which in turn contribute to developing mistrust among them. Figure 1 shows assumptions between people working in a software project. Whenever the assumptions do not match the reality, we become unsatisfied and conflict arises. Moe and Šmite [28] report the key factors that cause lack of trust.

As a solution to the problems, researchers suggest to take necessary action to mitigate the factors contributing to lack of trust [13, 7, 15, 26]. Others suggested considering a flexible and adaptable software development method that facilitates more communication and coordination among the team members [39].

Thus it can be argued that invalid assumptions may be a source of mistrust. Therefore, the solution should be directed toward where the problem originates. The idea of assumption management of human factors can also be applied, in which case it can build and maintain trust among the development teams. Thus it would be possible to reduce the key factors in a cost effective way that act as hindrances toward building and maintaining trust in the distributed working environment.

V. SUMMARY

Making assumptions is unavoidable when developing software systems. This article has provided an overview showing that assumptions are used in a number of different areas of software and system engineering. It is clear that there is a lack of integrated approaches toward systematic assumption management, enabling quantitative analysis and checks of assumptions, which would ultimately mitigate the key challenges associated with the assumptions. Mitigation of the challenges would support virtual integration of components, continuous deployment and more loosely coupled CPSs development. A holistic assumption management framework can offer different services to such other frameworks such as accessing the assumptions and their properties, on-request assumptions validation, on-request assumption updates, report errors, warnings, etc.

Currently, we are working on building a meta-model to capture assumptions at different system levels, e.g., component, subsystem, and system. Initially, we focus on the software and system architecture, with a particular focus on formal architecture specifications captured in an architecture modeling language such as AADL or OMG MARTE. The goal is to capture assumptions explicitly in the architecture model and conduct automated and quantitative analysis of the model. Thus part of the scope is to generate methods and tools for assumption-based verification and validation conducive to enabling smooth integration and continuous deployment of software systems.

REFERENCES

[1] A. Miranskyy, N. Madhavji, M. Davison, and M. Reesor, "Modelling assumptions and requirements in the context of project risk," in *Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on*, 2005, pp. 471-472.

[2] A. S. Tirumala, "An assumptions management framework for systems software," Ph.D. thesis. 2006.

[3] A. Tang, "A rationale-based model for architecture design reasoning," Ph.D. thesis. 2007.

[4] A. van Lamsweerde. Requirements engineering in the year 00: a research perspective. In ICSE '00: Proceedings of the 22nd international conference on Software engineering, pages 5–19. ACM Press, 2000.

[5] C. B. Haley, R. C. Laney, J. D. Moffett, and B. Nuseibeh, "Using trust assumptions with security requirements," *Requirements Engineering*, vol. 11, no. 2, pp. 138-51, 2006.

[6] C. B. Haley, R. Laney, J. D. Moffett, and B. Nuseibeh, "Security requirements engineering: A framework for representation and analysis," *IEEE Transactions on Software Engineering*, pp. 133–153, 2008.

[7] D. Bandow 2001. Time to create sound teamwork. *Journal for Quality and Participation* 24(2): 41.

[8] D. Garlan, R. Allen, and J. Ockerbloom, "Architectural mismatch: Why reuse is so hard," *Software, IEEE*, vol. 12, no. 6, pp. 17–26, 1995.

[9] D. Garlan, R. Allen, and J. Ockerbloom, "Architectural mismatch: why reuse is still so hard," *Software, IEEE*, vol. 26, no. 4, pp. 66–69, 2009.

[10] D. Hislop, Knowledge management in organizations: A critical introduction. Oxford University Press, 2005.

[11] D. Parnas. Information distribution aspects of design methodology. In *Proceedings of the 1971 IFIP Congress*, Amsterdam, November 1971. North Holland Publishing.

[12] D. Jackson, M. Thomas, and L.I. Millett, *Software for dependable systems: sufficient evidence?*, Natl Academy Pr, 2007.

[13] E. Rocco 1998. Trust Breaks Down in Electronic Contexts but Can be Repaired by Some Initial Face-to-face Contact. ACM: Los Angeles, CA, New York.

[14] G. A. Lewis, T. Mahatham, and L. Wrage, "Assumptions management in software development," 2004.

[15] G Piccoli, B. Ives 2003. Trust and the unintended effects of behavior control in virtual teams. *Mis Quarterly* 27(3): 365–395.

[16] H. Ordibehesht, "Explicating Critical Assumptions in Software Architectures Using AADL," *rapport nr.: Report/Department of Applied Information Technology 2010: 115*, 2010.

[17] Inquiry Board. ARIANE 5 – Flight 501 Failure. Inquiry Board report, http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf (March 2011)

[18] I. Ostacchini and M. Wermelinger, "Managing assumptions during agile development," in *2009 ICSE Workshop on Sharing and Reusing Architectural Knowledge (SHARK 2009), 16 May 2009*, Piscataway, NJ, USA, 2009, pp. 9-16.

[19] J. A. Dewar, Assumption-Based Planning; A Planning Tool for Very Uncertain Times. DTIC Document, 1993.

[20] J. Bosch, "Software Architecture: The Next Step," in *Software Architecture*, vol. 3047, F. Oquendo, B. C. Warboys, and R. Morrison, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 194-199.

[21] J. Tyree and A. Akerman, "Architecture decisions: demystifying architecture," *IEEE Software*, vol. 22, no. 2, pp. 19- 27, Apr. 2005.

[22] J. Viega, T. Kohno, and B. Potter, "Trust (and mistrust) in secure applications," *Communications of the ACM*, vol. 44, no. 2, pp. 31-6, Feb. 2001.

[23] K. Thompson, "Reflections on trusting trust," *Communications of the ACM*, vol. 27, no. 8, pp. 761–763, 1984.

[24] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[25] L. R. Cai, J. S. Bradbury, and J. Dingel, "Discovering Architectural Mismatch in Distributed Event-based Systems using Software Model Checking," 2006.

[26] M Ali Babar, JM Verner, PT. Nguyen. 2006. Establishing and maintaining trust in software outsourcing relationships: an empirical investigation. *Journal of Systems and Software* 80(9): 1438–1449.

[27] M. Spiegel, J. Reynolds, and D. C. Brogan, "A case study of model context for simulation composability and reusability," in *Proceedings of the 37th conference on Winter simulation*, Orlando, Florida, 2005, pp. 437–444.

[28] N. B. Moe and D. Šmite, "Understanding a lack of trust in Global Software Teams: a multiple-case study," *Software Process: Improvement and Practice*, vol. 13, no. 3, pp. 217–231, 2008.

[29] N.G. Leveson, "Role of software in spacecraft accidents," *Journal of spacecraft and Rockets*, vol. 41, 2004, pp. 564–575.

[30] P.-A. Quinones, S. R. Fussell, L. Soibelman, and B. Akinci, "Bridging the gap: discovering mental models in globally collaborative contexts," in *Proceeding of the 2009 international workshop on Intercultural collaboration*, Palo Alto, California, USA, 2009, pp. 101–110.

[31] P. Feiler, D. Cluch, J. Hudak. The Architecture Analysis & Design Language (AADL): An Introduction. CMU/SEI-2006-TN-011.

[32] P. Feiler, L. Wrage, and J. Hansson, "System Architecture Virtual Integration: A Case Study," Software Engineering Institute, CMU, Technical Report CMU/SEI-2009-TR-017, Nov. 2009.

[33] P. Lago and H. van Vliet, "Explicit assumptions enrich architectural models," in *27th International Conference on Software Engineering, 15-21 May 2005*, Piscataway, NJ, USA, 2005, pp. 206-14.

[34] P. J. Ågerfalk, B. Fitzgerald, H. Holmström, B. Lings, B. Lundell, and E. Ó. Conchúir, "A framework for considering opportunities and threats in distributed software development," in *Proceedings of International Workshop on Distributed Software Development*, France, 2005, pp. 47–61.

[35] P. Kruchten, P. Lago, and H. Vliet, "Building Up and Reasoning About Architectural Knowledge," in *Quality of Software Architectures*, vol. 4214, C. Hofmeister, I. Crnkovic, and R. Reussner, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 43-58.

[36] R. De Lemos, C. Gacek, and A. Romanovsky, "Architectural mismatch tolerance," pp. 175–194, 2003.

[37] R. Farenhorst and R. C. Boer, "Knowledge Management in Software Architecture: State of the Art," in *Software Architecture Knowledge Management*, M. Ali Babar, T. Dingsøyr, P. Lago, and H. Vliet, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 21-38.

[38] R. Roeller, P. Lago, and H. van Vliet, "Recovering architectural assumptions," *Journal of Systems and Software*, vol. 79, no. 4, pp. 552-573, Apr. 2006.

[39] R Sabherwal 1999. The role of trust in outsourced IS development projects. *Communications of the ACM* 42(2): 80–86.

[40] R. William P. et al., "Report of the Presidential Commission on the Space Shuttle Challenger Accident," U.S. Government Accounting Office, Washington, D.C., 1986.

[41] S. Fickas and M. S. Feather, "Requirements monitoring in dynamic environments," in *Requirements Engineering, 1995., Proceedings of the Second IEEE International Symposium on*, 1995, pp. 140-147.

[42] S. Uchitel and D. Yankelevich, "Enhancing architectural mismatch detection with assumptions," in Engineering of Computer Based Systems, 2000. (ECBS 2000) Proceedings. Seventh IEEE International Conference and Workshopon the, 2000, pp. 138-146.

[43] T. Dingsøyr and H. Vliet, "Introduction to Software Architecture and Knowledge Management," in *Software Architecture Knowledge Management*, M. Ali Babar, T. Dingsøyr, P. Lago, and H. Vliet, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 1-17.

[44] V. Page, M. Dixon, and I. Choudhury, "Security risk mitigation for information systems," *BT technology journal*, vol. 25, no. 1, pp. 118–127, 2007.