Multilevel Context-Aware Software Architecture Decision Framework
with Probabilistic Graphical Models

BY
PLAMEN P. PETROV
B.S., Technical University of Sofia 1989
M.S., Loyola University Chicago 1992
M.E.M., Northwestern University 2007

THESIS
Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2014

Chicago, Illinois

Defense Committee:
Ugo Buy, Chair and Advisor
Peter Nelson
Mark Grechanik
Houshang Darabi, Mechanical and Industrial Engineering
Robert L. Nord, Carnegie Mellon University
ACKNOWLEDGEMENTS

I would like to express my gratitude to my advisors – Ugo Buy and Robert L. Nord for their unwavering support, belief in my research and outstanding mentorship. Without their guidance and support I wouldn’t have been able to accomplish this significant milestone. I would also like to thank Peter C. Nelson both member of my thesis committee and my advisor when I first joined the program. He kindled my passion for Artificial Intelligence and continued to stay engaged, supporting and guiding me throughout the years. I would like to thank the rest of my thesis committee — Mark Grechanik and Houshang Darabi -- for their insights, support and assistance. I would also like to acknowledge Piotr Gmytrasiewicz, who was part of my proposal committee and made suggestions important to the conduct of the study.

I would like to express my deepest gratitude to my family that inspired, encouraged and supported me throughout the years.

PP
# TABLE OF CONTENTS

1 INTRODUCTION ........................................................................................................................................................................... 1

2 BACKGROUND ON ANALYSIS AND DESIGN OF SOFTWARE ARCHITECTURE .... 4

3 INTRODUCING CONTEXTUAL FACTORS AND ARCHITECTURAL EFFECTS........ 6

   3.1 CONTEXTUAL ENVIRONMENT AND CONTEXTUAL FACTORS .......... 7

   3.2 CONTEXTUAL ENVIRONMENT DIMENSIONS ........................................... 13

   3.3 CONTEXTUAL CONSTRAINTS AND CONTEXTUAL CONSIDERATIONS .... 18

   3.4 MACRO-ARCHITECTURAL PREFERENCES ............................................... 20

   3.5 MACRO-ARCHITECTURAL METRICS ...................................................... 24

       3.5.1 BACKGROUND ON PREFERENCE AND UTILITY THEORY ... 25

       3.5.2 METRICS AND MACRO-ARCHITECTURAL PREFERENCES ........ 26

   3.6 MULTILEVEL CONTEXT-AWARE ARCHITECTURE ................................... 29

   3.7 CONTEXTUAL FACTORS AND REQUIREMENTS ENGINEERING ............ 36

       3.7.1 Forward Inferred Macro-Architectural Requirements .................. 40

       3.7.2 Backward Inferred Macro-Architectural Requirements ................ 42

       3.7.3 REVERSE REQUIREMENTS ELICITATION PROCESS .................... 44

4 PROBABILISTIC MACRO-ARCHITECTURAL DECISION FRAMEWORK .......... 46

   4.1 MACRO-ARCHITECTURE AS DECISION ANALYSIS PROBLEM .......... 47

   4.2 PROBABILISTIC GRAPHICAL MODELS ................................................. 50
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.1</td>
<td>BAYESIAN NETWORKS AND DECISION NETWORKS</td>
<td>51</td>
</tr>
<tr>
<td>4.3</td>
<td>MODELING CONTEXTUAL FACTORS, PREFERENCES AND CAUSAL RELATIONSHIPS</td>
<td>53</td>
</tr>
<tr>
<td>5</td>
<td>CASE STUDIES AND RESULTS</td>
<td>57</td>
</tr>
<tr>
<td>5.1</td>
<td>SERVICE ORIENTED ARCHITECTURE DECISION</td>
<td>58</td>
</tr>
<tr>
<td>5.1.1</td>
<td>SCENARIO ANALYSIS</td>
<td>69</td>
</tr>
<tr>
<td>5.2</td>
<td>“OPEN SOURCE OR COTS” AND “DSL OR CONFIG FILE” DECISIONS</td>
<td>77</td>
</tr>
<tr>
<td>6</td>
<td>CONCLUSIONS AND FUTURE RESEARCH</td>
<td>82</td>
</tr>
<tr>
<td></td>
<td>REFERENCES</td>
<td>84</td>
</tr>
<tr>
<td></td>
<td>VITA</td>
<td>92</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 1: Contextual Environment and Software Architecture. ................................................... 11
Figure 2: Contextual Factors and Macro-Architectural Preferences. ........................................... 22
Figure 3: Macro-Architectural Metrics. ........................................................................................ 27
Figure 4: Multilevel Architecture. ............................................................................................... 30
Figure 5: Diagram of the Macro-Architectural Level. ................................................................. 31
Figure 6: Multilevel Context-Aware Architecture. ....................................................................... 32
Figure 7: Macro-Architectural Requirements. ............................................................................ 39
Figure 8: Decision Hierarchy Model. .......................................................................................... 48
Figure 9: Architecture Decision Hierarchy Model. ................................................................. 49
Figure 10: Decision Network Implementation. ......................................................................... 61
Figure 11: CPT for System Reuse. ............................................................................................ 64
Figure 12: CPT for Development Method. ................................................................................. 65
Figure 13: CPT for SOA Clients Available. ................................................................................ 66
Figure 14: Utility Table for Affordability. ................................................................................ 66
Figure 15: Utility Table for Simplicity. .................................................................................... 67
Figure 16: Utility Table for Customer Satisfaction. ............................................................. 68
Figure 17: Compiled Decision Network. ................................................................................ 69
Figure 18: Scenario 1 Decision Network. ................................................................................ 70
Figure 19: Scenario 2 Decision Network. ................................................................................ 71
Figure 20: Scenario 3 Decision Network. ................................................................................ 72
Figure 21: Scenario 4 Decision Network. ................................................................................ 73
LIST OF FIGURES (continued)

Figure 22: New System Reuse Likelihood Evidence. .............................................................. 74

Figure 23: Scenario 5 Decision Network.................................................................................. 75

Figure 24: "Open Source - COTS" and "DSL - Config File" Decision Model ......................... 78
SUMMARY

The thesis describes (1) our research in the area of software architecture, (2) the problems with the current practices that we have identified, (3) the inventions we have made to solve those problems, (4) the empirical case studies we have made to validate our approach, (5) the tools and supporting models we have developed to enable our solution, (6) insights and conclusions from our research and experiments, and (7) suggestions to extend our research in the future. The problem we researched is: why do otherwise technically sound software architecture decisions and designs fail to achieve the expected outcomes and prove to be inappropriate? Our breakthrough insight is that contextual environment factors that are not associated with software architecture and are easily overlooked by software architects are the culprit. We also realized that those contextual factors are not easily captured as requirements, and are not easily modeled using existing structural decomposition engineering design methods. First we split out the contextual analysis phase as a separate macro-architectural level. Then we looked at decision analysis for ideas and introduced an innovative approach reframing the contextual environment analysis as a decision theory problem. Then we modeled it by adapting methods and tools from decision analysis, more specifically developing a Probabilistic Macro-Architecture Decision Framework based on Bayesian Networks (BN) and Decision Networks (DN) to handle the inherent uncertainty and complexity of the problem [126][132]. We developed several working models using a Bayesian Networks tool that enabled us to perform empirical validation of our research. We reported on the outcomes of the validation case studies that we performed. Finally we recommended future extensions to our research based on machine learning and data mining techniques to further contribute to the theory and practice of software architecture.
1 INTRODUCTION

This thesis summarizes the software architecture research and presents the inventions and contributions that we made to the software architecture theory and practice as part of our enrollment in the Ph.D. program at the University of Illinois at Chicago. The work reported here was performed exclusively by the author of this document under the supervision of the dissertation committee. Our interest in this research was triggered by problems that we encountered and insights that we made during our work as software architecture practitioners. Those insights triggered an in-depth analysis of the problem space, rigorous research and experimentation and development of new architectural decision framework and supporting tools to address the original problems.

Software development is a critical activity in the modern society that consumes a lot of resources and produces a lot of benefits. Software architecture analysis and design is among the methods and techniques that evolved over time to enable efficient and cost effective development of high quality software systems. The software industry however continues to struggle with cost overruns, major system reworks, delivery delays, quality issues and even straight project failures. There is a lot of potential for improvement.

One such area of concern and potential for improvement is the software architecture design process. Some major improvements were made over the last decades, yet we observed ongoing problems with selecting the appropriate architecture and making the optional architectural decision. The problem that puzzled us and fascinated us at the same time was – why do technically sound software architecture decisions and designs prove to be inappropriate and
misaligned? This is a recurring problem and the proposed solutions did not seem to show significant improvements.

Software architecture theory and practice were focusing on improving the completeness of the requirements engineering processes and on increasing the rigor and detail of the structural decomposition modeling and design processes. Those efforts did not result in meaningful improvement to the software architecture quality and the system development outcomes.

As part of our research we made a breakthrough realization that some of the important forces driving the selection of the appropriate software architecture were contextual to the software system development, operational, financial, cultural, business and organizational environment and not directly tied to the purpose of the system and its requirements. Those contextual factors were easily overlooked and ignored by software architects as they were considered to be applicable to project managers, finance professionals, senior business managers, but not to software architects. We went on a journey to understand how these contextual factors impact the software architecture of a system, and then to invent a modeling and decision framework to incorporate them in the software architecture design process.

During our research journey we identified an opportunity to reframe some of the software architecture processes as decision analysis problems. This insight allowed us to apply powerful formalisms from Decision Analysis and Artificial Intelligence to the software architecture domain and resulted in some excellent outcomes. The Probabilistic Macro-Architectural Decision Framework formally described in this thesis is a concrete implementation of the theoretical concepts that we introduced. We validated our research by studying several cases
from industry and applying the new framework and techniques to empirically compare the outcomes to the predictions. As part of our research we developed supporting models based on the Netica Bayesian Network (BN) tool from Norsys[127]. The Netica models and implementation are made available freely as requested.
2 BACKGROUND ON ANALYSIS AND DESIGN OF SOFTWARE ARCHITECTURE

There are many definitions of software architecture, but we will use the definition from [102]

“The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both”

Another definition that captures well the essence of software architecture is the following: “The architecture of a software-reliant system identifies the main components in the overall system and the interactions among those components” [37].

The software architecture for a system is a key milestone in the software development process. Architectural decisions have a significant effect not only on the structure and functionality of each component in the system being developed, but also on the success or failure of the overall software project. To date the decisions giving rise to a given architecture for a software system are typically based on the software requirements, especially functional and non-functional requirements.

A good summary of 5 software architecture industrial approaches can be found in [5]. The theory and practice of software architecture analysis and design has evolved over the years with the development of several theories and methodologies.

Software architecture from its beginning as a discipline has focused on the structure of the system and the interactions among its structural parts, identified repeating patterns of structuring the system (architectural styles), and studied the system properties that ensued from those different structural patterns. [10] The initial research was somewhat internally focused on the properties that different architecture styles exhibit.
Software architecture researchers had introduced the concept of multiple views to capture the concerns of different stakeholders. The 4+1 View model is a good representative of the dominant approach to software architecture from the mid 1990’s through the end of the century [109]. The major architectural drivers are functional requirements captured as use cases in the “logical” view. The “process” view adds some non-functional requirements. The “development” and “physical” views capture some development and system non-functional requirements. The “4+1” model focuses on scenarios (select use cases) that capture the systems major functionality and how the views work together.

Software architecture researchers and practitioners realized over time that non-functional requirements play a much larger role than functional requirements in defining the appropriate software architecture of a system. The Attribute-Driven Design (ADD) method from the Software Engineering Institute (SEI) at Carnegie Mellon University in Pittsburgh, Pennsylvania focused on quality attributes (i.e., non-functional requirements) as the dominant driver for designing the software architecture of a system [23]. The complementary Quality Attribute Workshop (QAW) method from SEI provided a technique to elicit and capture quality attributes [85][86]. Similarly the Siemens Four-Views (S4V) method relied on four views to capture different engineering concerns, but also introduced a recurring Global Analysis activity to address organizational, technological and product factors [12][38].
3 INTRODUCING CONTEXTUAL FACTORS AND ARCHITECTURAL EFFECTS

As part of our research we introduced the concepts of contextual environment and contextual factors for software architecture.

The theory and practice of software architecture methods have successfully developed and evolved over the years bringing a level of rigor, quality and predictability to the development of software systems. The focus has been identifying systemic methods and techniques that would enable the selection of the optimal software architecture for a certain purpose. Researchers argued successfully that the selected software architecture is not inherently a bad architecture or a good architecture, but should be measured as to how well it “fits for the purpose” it is intended. [102]. We largely agree with this argument, but our research has also demonstrated that “fitness for purpose” is a necessary, but not sufficient prerequisite for the appropriateness of a software architecture. We have observed that otherwise appropriate software architecture for its purpose may still fail if it does not fit the environment in which it is designed, developed, implemented, deployed, operated, funded, promoted, marketed, staffed, and licensed.

We call this the contextual environment of the software architecture and software system and we call contextual factors the relevant elements that constitute the contextual environment. We argue that software architecture should be measured not just how well it is “fit for purpose”, but also how well it is “fit to context”.

6
3.1 CONTEXTUAL ENVIRONMENT AND CONTEXTUAL FACTORS

Let’s more formally define contextual factors and contextual environment.

- We define contextual factors as strategic/business, organizational, financial, operational, and cultural features of the surrounding environment within which the software system is designed, developed, implemented and operated. Contextual factors are the characteristic features of the surrounding environment that are not explicitly specified as functional requirements, non-functional requirements or business goals for the system, but still have direct or indirect effect on the software architecture and as a consequence on the success or failure of the software system.

- The collection of the contextual factors constitutes the contextual environment of the system.

We would like to emphasize two important points:

- In our research contextual factors are elements of the environment that are external to the immediate system requirements and that have not been traditionally captured as either functional requirements or quality attributes (i.e., non-functional requirements)

- In our research we study how the contextual factors impact and should influence the selection of the appropriate software architecture and not how the contextual environment affects in general the success or failure of a certain business or project

Current practices rely on the assumption that all factors affecting architectural decisions were appropriately captured. In reality, this assumption often does not hold; architecture decisions are based not only on explicitly-stated requirements but also on unspoken requirements and implicit
knowledge that are part of the contextual environment. This environment embodies enterprise and system concerns such as the structure of the development organization(s), development funding models, deployment strategies, and software evolution. It is generally understood that ignoring contextual factors when making architectural decisions can lead to project failure.

The introduction of the concepts of contextual factors and contextual environment aligns with the overall evolution of the theory and practice of software architecture over the years. The trend has continuously been to move from more internally focused concerns to greater external awareness. The trend has also been to move from more concrete and functionally focused software architectural drivers to more abstract and non-functional considerations.

Software architecture researchers and practitioners realized over time that non-functional requirements play a much larger role than functional requirements in defining the appropriate software architecture of a system. The Attribute-Driven Design (ADD) method from SEI focused on quality attributes (i.e., non-functional requirements) as the dominant driver for designing the software architecture of a system [23]. The complementary Quality Attribute Workshop (QAW) method from SEI provided a technique to elicit and capture quality attributes [85][86]. Similarly the Siemens Four-Views (S4V) method relied on four views to capture different engineering concerns, but also introduced a recurring Global Analysis activity to address organizational, technological and product factors [12][38][52].

The shift in focus from functional to non-functional requirements as a driver for software architecture definition was a major development, but it continued to evolve over time. Researchers at SEI observed that software architects focus most on functional requirements that contribute least to the software architecture instead of quality attributes, but also that even the
best requirements specification rarely capture what is most useful for the software architecture. In addition, the organizational business goals should somehow be incorporated as drivers [89][125]. The researchers at the SEI introduced the Pedigreed Attribute eLicitation Method (PALM) as a technique to elicit and capture business goals as quality attributes.

The introduction of contextual factors and contextual environment as part of our research is yet another evolution along the same trend of making software architecture more contextually relevant. Unlike the PALM method and BAPO agile business value that focused on identifying business goals and capturing them as quality attribute requirements, we observed that a large number of architecturally significant drivers are not explicit business goals, but are rather implicit constraints and considerations that influence the software system and its architecture with a certain probability [113]. This insight allowed us to reframe the early part of the architecture process as a higher-level of abstraction, probabilistic decision analysis problem rather than the traditional more concrete and deterministic, structural decomposition and engineering problem. This breakthrough led us to the introduction of the Probabilistic Macro-Architectural Decision Framework (PMADF) that adapts the Bayesian Networks (BN) and Decision Networks formalisms as a more appropriate decision analysis-based method for the early stage of software architecture analysis and design [83][132].

Our original research work introducing the concept and technique of “contextualization” of software architecture. In addition, elevating the importance of “fit to context” architecture in addition to “fit for purpose” aligns with a trend across the software architecture community. Other research groups have also recognized the significance of context in defining successful software architecture. A notable acknowledgement of the significance of context for software architecture is the introduction of an entire chapter on context in the 3rd edition of the “Software
Architecture in Practice’ book from 2013 [102]—a book considered by many as the ultimate body of knowledge on software architecture. The earlier 2nd edition from 2003 barely mentioned context as a consideration for software architecture, while the 3rd edition from 2013 dedicated an entire chapter to the topic and discussed it extensively.

Our research over the last few years has focused extensively on understanding context in software architecture and inventing effective techniques for incorporating it as a major software architecture driver. We take a distinctly different approach in how we treat and handle context—we view it and treat it as a “first class citizen” when it comes to the analysis and design of software architecture. In contrast with Clements and Bass [89][125] who extended the existing Quality Attributes framework to capture business goals as quality attributes we modeled contextual factors as constraints that influence the software architecture of the system with a certain probability. We observed that many contextual factors are not explicitly documented and frequently not consciously defined as goals within the organization and should be rather modeled as constraints. At a conceptual level, the contextual environment is a set of constraints and considerations that exist without necessarily a deliberate design and that permeate the software system’s development and sponsoring organizations at many levels across its business processes and across strategic, operational, organizational, financial, and cultural domains. The following diagram provides a high level view of these concepts.
Although different factors can impact the success or failure of a business and of a software system, we are only interested in factors that have a direct or indirect effect on the software architecture of the system under review or design. By definition, these are the contextual factors for the software system and the software architecture. Other factors may be of interest to economists, project managers, financial professionals, human resources professionals, among others. However, if these factors do not affect the software architecture of a system as architectural drivers they are out of scope for our research. The existence of a causal relationship between a contextual factor and the suitability of selecting a software architecture is required by definition. We did an extensive research of the business, organizational, financial and engineering domains and identified the five dimensions shown in Figure 1 as the optimal

---

**Figure 1: Contextual Environment and Software Architecture.**
categorization of the contextual environment. We cover all key contextual aspects and can consistently assign each contextual factor to one of these dimensions.

Contextual factors can be easily abused and overextended to cover a large number of the existing architectural drivers. We can envision a setup where all architectural drivers – be those functional requirements, quality attributes or business goals – are modeled as contextual factors. This is clearly not the purpose of our framework; this approach would be both inappropriate and ineffective. The contextual factors framework complements and extends the traditional architectural drivers and does not replace or supersede them. The key differentiator here is that functional requirements, quality attributes and business goals are intentional with respect to the software system and the business scenario it supports. They are typically documented either in a business document, technical or requirements specification. This is in contrast to contextual factors, which are unintentional with respect to the software system and business scenario under consideration. Contextual factors exist regardless of the software system or business scenario. These factors happen to impact the software architecture of the system in a way that is frequently overlooked by the system’s stakeholders.
3.2 CONTEXTUAL ENVIRONMENT DIMENSIONS

After extensively studying literature and prior art, we developed a categorization of contextual factors along several contextual dimensions. This categorization is intended to make it easy for a software architect to identify contextual factors when modeling a specific scenario.

We define the following dimensions of the contextual environment:

- Strategic/business
- Organizational
- Financial
- Operational
- Cultural

Follow some examples of contextual factors within these dimensions. [22]

- Strategic/business
  - Business Model Positioning
    - Low cost commodity business focus
    - High-margin differentiated business
  - Target market size
    - High volume
    - Low volume niche
  - Business lifecycle phase
    - Growth business
    - Stable “cash flow” sustaining business
    - Declining business
  - Software development role
✓ Revenue and profit generating core business role
✓ Cost center supporting role

- Organizational
  - Development Staff Sourcing Model
    ✓ On staff employees
    ✓ Contracted augmentation staff
    ✓ Outsourced consulting
  - Development Staff Skill Level
    ✓ Experienced
    ✓ Inexperienced
  - Development Staff Experience Focus
    ✓ Large scale new software development
    ✓ Existing software maintenance
    ✓ Packaged software integration

- Financial
  - Software system funding model
    ✓ Enterprise-wide pre-allocated
    ✓ Line-of-business charge back
  - Funding Source
    ✓ Product marketing profit center
    ✓ Department cost center
  - Funding horizon
    ✓ Specific short term feature funding
    ✓ Multi-year multi-release funding
  - Funding relative priority
    ✓ Minimize cost as primary consideration
    ✓ Deliver as specified with cost as secondary consideration
• Operational
  ➢ Deployment sourcing
    ✓ Internal data center
    ✓ Outsourced data center
  ➢ Software maintenance
    ✓ Internal team
    ✓ Outsourced team
  ➢ Hardware diversity support
    ✓ Single target platform
    ✓ Few target platforms
    ✓ Numerous target platforms
  ➢ Deployment instances
    ✓ Single system deployment
    ✓ Few instances deployment
    ✓ Large number of customer deployments

• Cultural
  ➢ Risk profile
    ✓ Encourages risk taking
    ✓ Tolerates risk
    ✓ Risk averse
  ➢ Cost profile
    ✓ Deliver on projected budget is highest priority
    ✓ Deliver with aggressive low cost highest priority
  ➢ Timeline profile
    ✓ Deliver on projected timeline is highest priority
    ✓ Deliver with aggressive short timeline is highest priority
  ➢ Feature delivery profile
Deliver on projected features is highest priority
Deliver with aggressive number of features is highest priority

Quality profile
Deliver on projected quality metrics is highest priority
Deliver with ability to remediate quality issues is highest priority

Cost–timelines-features-quality profile
Cost is primary predisposition
Meet timeline is primary predisposition
Deliver projected features is primary predisposition
Meet high quality standard is primary predisposition

The contextual factors within the different contextual environment dimensions serve as influencers and constraints to the software architecture. Some of those may be overridden as explicit functional requirements or quality attributes for a specific software system scenario, but otherwise they happen to be part of the implicit environment within which the software system resides.

An example for a strategic business contextual factor and the architectural influence would be the following scenario:

Target market size
High volume
Low volume niche

A high volume target market size would emphasize architectural decisions that favor maintainability, since updates to many instances of the software system need to be economically maintained. Similarly the high volume target market size would deemphasize customization as
architectural preference, since commoditized large volume implementations would not target very specific customizations.

On the other side, a low volume niche market would emphasize customizable architectural decisions and relatively deemphasize maintainability for the same reasons, but in reverse.

As we discuss in a later chapter, we define macro-architectural effects as the architectural considerations and decision that are influenced by contextual factors. Evidently, there is a causal relationship between one or more contextual factors and an architectural effect.

Contextual factors are probabilistic in nature as there is inherent uncertainty associated with them being correct. In line with the Bayesian or evidential interpretation of probability that we take across our research and present formally in a later section, we assign a prior probability to the contextual factors and allow for that probability to be updated as new relevant evidence becomes available.
3.3 CONTEXTUAL CONSTRAINTS AND CONTEXTUAL CONSIDERATIONS

Depending on the level of uncertainty and the ability of the software architect to influence a contextual factor, we categorize contextual factors as **contextual constraints** or **contextual considerations**. [83]

**Contextual constraints** are factors taken for given—these are externally controlled factors that are considered non-negotiable with a very high probability that they will not change. We consider the probability that they will not change to be 1.

**Contextual considerations** are external factors and generally outside the architect’s immediate influence; however, they are not deterministically fixed. The probability that they will not change is less than 1. The probability of them staying as projected or changing is a value that can be predicted by the architect based on expert-level knowledge.

Contextual constraints may have a causal effect on contextual considerations. Both contextual constraints and contextual considerations have a causal effect on software architecture.

A certain contextual factor may manifest itself as a contextual constraint in one software development scenario and as a contextual consideration in another. When the software architect is performing the contextual environment analysis for a specific software system, he would assess the probability of each relevant contextual factor and determine if it should be classified as a constraint or consideration.

The following example illustrates this point. Two potentially relevant contextual factors from the financial dimension are “funding source” and “funding horizon”. The funding source can be “product marketing profit center” or “department cost center” and the software architect should identify that as part of performing contextual environment analysis. In addition, the software
architect should estimate the probability that this contextual factor stays at that value. As we describe later, the causal relationship between the “funding source” contextual factor and an architectural effect such as selecting Service Oriented Architecture (SOA) [39] as an architectural style depends not only on the value of the contextual factor, but also on the probability of that contextual factor staying the same or changing over time. Similarly, as part of the contextual environment assessment the software architect may assess that the “funding horizon” for the software system is either “specific short term feature funding” or “multi-year multi-release funding”. The software architect would also estimate the probability of the “funding horizon” changing over time. If the software architect assesses that the “funding source” is “department cost center” and that there is a very low or negligible probability that this would change over time, the “funding source” contextual factor would be modeled as a contextual constraint. Similarly, the software architect may uncover that the “funding horizon” for the system is “specific short term feature funding” and estimate the probability of this factor staying that way to be 0.75. The “funding horizon” contextual factor would be modeled as a contextual consideration with a value of “specific short term feature funding” and with prior probability of 0.75. It is possible that there is a causal relationship between contextual factors as one factor may influence the conditional probability of another. Conditional probability is only possible and relevant for contextual considerations. Contextual constraints already have a prior probability of 1 and cannot be influenced by other contextual factors.
3.4 MACRO-ARCHITECTURAL PREFERENCES

We define macro-architectural preferences as the architectural considerations and decision that are influenced by contextual factors and the causal effects of the contextual environment. By definition there is a causal relationship between one or more contextual factors and their architectural effects that results in a bias toward a given architectural decision – the macro-architectural preference. The macro-architectural preference is appropriately named “preference” and not “decision” as it is both probabilistic and subjective with respect to a certain utility value assigned by the system stakeholders. The Probabilistic Macro-Architectural Decision Framework (PMADF) that we define in a later chapter models the cause and effect relationship between contextual factors and macro-architectural preferences. The framework allows us to reason about the architectural effects and to perform scenario analysis. [83]

An example of a macro-architectural preference is selecting a Service Oriented Architecture (SOA) as the architectural style for a system under design and the probability that such an architectural decision will result in a successful system implementation considering the causal effects of relevant contextual factors. Macro-architectural preferences are influenced by contextual factors with a certain prior conditional probability, but ultimately the software architect has to make the architectural decision – as in this case whether to select Service Oriented Architecture or not. These architectural decisions and enabling the software architect to select the choice that is more likely to result in a successful implementation is at the core of the Probabilistic Macro-Architectural Decision Framework (PMADF).

The causal relationship between contextual factors and macro-architectural preferences is explicitly identified and encoded as a conditional probability extracted either from historical
projects probability distributions or through knowledge engineering elicitation from subject matter experts.

In the example from the previous section, both “funding horizon” and “funding source” have a causal relationship with the Service Oriented Architecture (SOA) macro-architectural preference. The “funding source” contextual factor can be either “product marketing profit center” or “department cost center”. By reviewing historical project cases and eliciting domain knowledge from subject matter experts through knowledge engineering techniques, we quantified the conditional probability for Service Oriented Architecture (SOA) to result in a successful implementation [83]. The rationale behind that causal relationship is that a “product marketing profit center” funding source is more likely to invest in the more sophisticated, more rigorous and more expensive “software product” development method that brings the discipline and techniques necessary to ensure a successful SOA implementation. A “cost center” funding source is much more likely to avoid the extra overhead and costs of a “software product” development method and to favor an “internal system” or “utility tool” development method. Those two development methods have a much lower likelihood of successful SOA implementation as observed from historical project study and validated through knowledge engineering elicitation from software architecture subject matter experts.

The causal relationship between the “funding source” contextual factor and the Service Oriented Architecture (SOA) macro-architectural preference has been identified as indirect with “development method” as the contextual factor that sits on the causal path between cause and effect. The following diagram visually shows the causal relationships.
In this specific software system example, the funding source happens to be preset with a probability of 1 that it will stay the same over time. Intuitively, this is understandable since if the product marketing department funds the development of a new product for external customers, it is unlikely that the funding will change to internal cost center. The development method on the other hand has a probability other than 1 as there is uncertainty in making that selection. The selection of the development method is correlated with the funding source and the probability if development method selection is encoded as a conditional probability.

The causal relationship between the development method contextual consideration and the “Use Service Oriented Architecture (SOA)” macro-architectural preference expresses the probability of a successful SOA implementation given that the development method is “software product”, “internal system” or “utility tool”. In addition to this probability of success, the macro-architectural preference
incorporates the utility assigned to the different possible decisions as described in the next chapter.
3.5 MACRO-ARCHITECTURAL METRICS

Macro-architectural preferences are influenced by contextual constraints and considerations, but ultimately the software architect has to make the macro-architectural decision taking into consideration the probability of success given the contextual factors. The likelihood of system implementation success is a good guidance on which architectural decisions are more attractive; however, it is not sufficient to quantify the quality of the architectural decision. We introduce macro-architectural metrics as the mechanism to quantify in a systemic way the quality and appropriateness of the architectural decision.

To enable comprehensive and optimal decisions, we include both objective “hard” measures that quantify the quality of the architecture, as well we subjective “soft” utilities that capture the desirability of architectural outcomes from the perspective of the key stakeholders. We categorize these two types of macro-architectural metrics as:

- objective measures, and
- subjective utilities

The objective measures include the following key non-functional parameters by which stakeholders assess the value of a software system (and implicitly its architecture):

- Cost (e.g., cost to build, cost to operate, total cost of ownership)
  - Measured in units of cost, such as dollars
- Time (e.g., time to market, time to build, etc.)
  - Measured in units of time, such as days
- Quality (e.g., discovered defects over a fixed period of time after system implementation)
  - Measured in units of work, such as number of defects, number of patches, etc.
The subjective utilities are harder to quantify, elicit and normalize, yet they are just as important for the evaluation of the software system and its architecture.

3.5.1 BACKGROUND ON PREFERENCE AND UTILITY THEORY

We rely on preference and utility theory to elicit and capture the subjective utility that stakeholders associate with a certain architectural decision [128][129]. Utilities capture the subjective value or “worthiness” of one option vs. another option to a stakeholder. The utility is represented by a certain value for a certain outcome or decision. In a continuous domain, a utility would be represented as a function – e.g., how much is winning a lottery prize from $0 to $500,000 million worth for you. In a discrete domain the utility would be represented by discrete values assigned to different possible outcomes – e.g. winning a car, a trip to Las Vegas, or a coffee maker machine. Utility incorporates the concept of value to a certain person or group of people; it is typically not identical to different people and not linear as it reflects the stakeholder’s subjective preferences. As an example, for a billionaire like Bill Gates winning $10 million lottery prize probably has a much lower utility that for an unemployed person with a large family to care for. Utility is thus not necessarily quantified as a dollar value, but rather as a number assigned to express how much the object is worth to the stakeholders. Utility also expresses the risk attitude of the stakeholder, thus reflecting much more precisely the values assigned to different outcomes under uncertainty than an alternative measure such as expected value. As an example if there is 50% probability of success of a project valued at $40 million, the expected value would be $20 million; however, considering the natural risk aversion of humans, the utility assigned to a project with such risk profile would be much lower. This difference between utility and expected value is called the risk penalty.
Utilities are part of active research and practice across numerous disciplines, including decision analysis. In our research we adapt the utility formalism to software architecture as an appropriate mechanism to capture the inherently uncertain and subjective preferences that play a major role in the decision making that software architects perform. In the Probabilistic Macro- Architectural Decision Framework, utilities serve to capture the highly subjective yet highly influential valuations of the different architectural options that are intrinsic to the human stakeholders of the system.

Measuring utilities is known to be a hard problem as they are usually not observable directly and need to be elicited through multiple indirect techniques. Utility elicitation techniques are part of active research and we have surveyed multiple techniques as part of our investigations [128][62][72]. A good survey of the different methods can be found at [108]. We have experimented with and used both the Simple Multi-Attribute Rating Technique (SMART) and the Analytical Hierarchy Process (AHP) techniques to elicit and capture utilities for macro-architectural decisions.

3.5.2 METRICS AND MACRO-ARCHITECTURAL PREFERENCES

It is the primary responsibility of the software architect to reliably quantify the objective measures of possible macro-architectural decisions, as well as to elicit the utility of the different possible macro-architectural decisions to the key stakeholders. The Probabilistic Macro-Architectural Decision Framework calculates the macro-architectural preferences of the possible architectural decisions. The contextual factors – constraints and considerations – provide the probabilistic framework to quantify the desirability of an architectural decision based on causal
relationship. The macro-architectural metrics – measures and utilities – provide the valuation framework for discovering how much different options are worth to the stakeholders. As we shall see in the following section, we integrate these concepts into the powerful modeling and inference apparatus of Probabilistic Graphical Models – Bayesian Networks and influence diagrams – to provide a tool for software architects to analyze architectural options, exercise different scenarios and make optimal decisions [126]. The following figure graphically shows the different concepts and how they are related:

![Diagram](image)

**Figure 3: Macro-Architectural Metrics.**

This is an extension to the example from the previous section - the causal relationship between the “funding source” contextual constraint, the “development method” contextual consideration and the “Use SOA” macro-architectural preference expresses the probability of a successful SOA
implementation given that the development method is “software product”, “internal system” or “utility tool”. In addition to this probability of success, the macro-architectural preference incorporates the “system cost” macro-architectural metric to capture the difference in cost associated with selecting a SOA architectural style or not. The combination of both inputs quantifies the value of each architectural decision in this scenario; the most valuable decision will be provided as the macro-architectural preference to the software architect.
3.6 MULTILEVEL CONTEXT-AWARE ARCHITECTURE

In the previous sections we made the case and developed the formalism to model contextual factors and their influence on architectural decisions.

We will now describe more formally the concept of multi-level architecture that we introduced in [22] and [113].

The rationale for the introduction of this approach lies in the following insights:

- Contextual factors are typically not considered during software architecture development.

- Contextual factors influence key decisions that are made early on by business stakeholders without considering their architectural impact.

- Different modeling techniques are more appropriate for different phases of software architecture development
  
  - Early phase analyzing contextual factors is more effectively modeled as decision analysis problem

  - Later phases are better modeled with traditional structural decomposition engineering methods

We split the software architecture process into two major levels:

- Macro-architecture - a higher abstraction level that models the contextual environment and the architectural preferences influenced by the contextual factors
- Micro-architecture - a lower abstraction, more concrete level that performs the conventional architecture analysis and design, but contextualized within the architectural preferences established by the macro-architecture.

The macro-architectural approach reframes the architectural process as a decision analysis problem and creates a probabilistic decision analysis framework based on probabilistic graphical models.

The micro-architecture employs the conventional structural decomposition engineering methods of software analysis and design (e.g. ADD, 4+1, etc.), but modified to incorporate the contextualization based on the architectural preferences specified by the macro-architecture.

The following diagram captures graphically the multi-level architecture concept:

**Figure 4: Multilevel Architecture.**
The macro-architectural level integrates in a probabilistic causal graphical model the contextual factors – constraints and considerations – with the macro-architectural metrics – measures and
utilities – to calculate the architectural preferences that contextualize and influence the micro-architectural design process.

The following diagram captures graphically the concepts and relationships in the macro-architectural level.

---

**Figure 5: Diagram of the Macro-Architectural Level.**

As we describe in later chapters we use the apparatus of Probabilistic Graphical Models and more specifically Bayesian Networks (BN) and Influence Diagrams to formally model the entities, influences, causal relationships, probabilities and utilities that capture the architectural context and allow reasoning and scenario analysis.
The following figure graphically describes the multilevel context-aware software architecture analysis and design process.

![Diagram of Multilevel Context-Aware Architecture](image)

**Figure 6: Multilevel Context-Aware Architecture.**

The top arrow describes the traditional flow of functional requirements, quality attribute requirements and design constraints to the definition of a “fit for purpose” software architecture,
called micro-architecture. The bottom arrow expresses the impact that the contextual environment has on the selection and definition of a “fit to context” software architecture, called macro-architecture. Global analysis (part of the Siemens Four Views architecture design approach [12]) is an approach to expand the requirements engineering process to cover concerns broader than the immediate functional and quality attribute requirements [52], [40]. We build on top of the global analysis approach to formalize the concept of contextual environment concerns that additionally cover some implicit sources of information to discover architectural requirements.

Based on architectural best practices, experience and rules of thumb, it can be practical to translate certain enterprise-level and system-level contextual environment concerns directly into macro-architectural heuristics, decisions and patterns, bypassing the step of translating the contextual environment into descriptive requirements. The traceability and proper documentation of the decisions is handled through the rationale section in the architectural heuristics and decisions artifacts [50][120][19][35].

**Example 1**—Suppose that the enterprise-level contextual environment includes a development organization committed to innovation whereby the organization is composed of several geographically distributed teams each specializing in a certain functional domain and technology. Some core functional domains are funded as an allocation across the entire organization and managed as a portfolio through a rigorous development lifecycle. Others are funded as a direct charge back to the specific project with target profit and loss (P&L) and return on investment (ROI) guidelines and managed through lightweight agile processes. In this case, an architecture with a good chance of success should account for those contextual environment concerns by selecting macro-architectural patterns and heuristics that fit the context.
The context-aware macro-architecture guides the solution structure to reflect the structure of the development organization and partitions the functionality to reflect the specialization of the various units. Additionally, functionality that benefits the entire enterprise is partitioned and assigned to the shared services development unit supported through the allocation-based funding model. Since the organization is committed to innovation, high-risk and leading-edge capabilities could be assigned to the agile development units to avoid unnecessary overhead in a fluid environment. Alternatively, these capabilities could be assigned to the units that allow for charge back of high-risk tasks with a probability of failure in their funding model.

Since the organization is both geographically distributed and committed to innovation, the architectural partitioning emphasizes loose coupling to allow for frequent changes in the innovative modules, as well as simple, but rigorous interface contracts among modules to avoid integration challenges and undesired dependencies. All these considerations have a profound effect on the selection of the appropriate software architecture and the ultimate success of the project.

Notice that these considerations, related to the contextual environment, are rarely discussed or captured as requirements in the current requirements engineering and software architecture practices; they would be quite hard to capture as explicit requirements. These considerations involve implicit knowledge, architectural rules of thumb, and practical experience. Yet an experienced and knowledgeable architect would likely identify those patterns and heuristics. These macro-architectural decisions should be captured as rationale and justifications in the architectural artifacts to allow for traceability, reasoning on the appropriateness of the decisions, and further tradeoff analysis.
Our approach bears some similarities to goal-oriented requirements analysis [54][56][44][60], and the evolution from a task-oriented approach [13]. Moreover, our approach expands on existing work to relate business goals to architectural requirements and to software architecture [89][125]. Traditional task and problem-oriented requirement analysis methods focus on what features and qualities the system under design should support. Goal-oriented analysis broadens the process to include the motivation and rationale of why the system should be constructed. We expand on these concepts by adding implicit enterprise-wide and system-wide contextual environment considerations to the requirements process that may not be viewed as explicit goals for the specific project, but have just as profound and significant impact on determining the appropriate architecture and design.
3.7 CONTEXTUAL FACTORS AND REQUIREMENTS ENGINEERING

Practitioners and researchers know well that software architecture processes and requirements engineering processes are closely interrelated. Traditional software architecture methods rely on functional requirements and quality attributes that are typically elicited through requirements engineering techniques. We introduced requirements engineering techniques to support contextual factors, macro-architectural preferences and multi-level architecture [40].

Current software architecture methods and processes generally rely on requirements being elicited and provided as an input to the selection of the appropriate software architecture [37][99][101][102][103]. The requirements engineering process typically includes numerous practices and steps (e.g., requirements elicitation, requirements analysis, negotiation, requirements specification, system modeling, requirements validation, requirements management) [90][91][92]. A first assumption is that stakeholder concerns can be elicited and explicitly captured as requirements.

There are many software development methods and processes, some being mostly sequential in nature (e.g., waterfall), others being iterative, incremental, spiral, and agile. Most generally accepted and widely adopted requirements engineering techniques rely on a second assumption that the flow of authoritative information and control is from the problem domain to the solution domain, that is, from stakeholder requirements to system requirements, architecture and design [92]. The second assumption does not preclude iterations between requirements and architecture, but rather speaks to the higher level of information authority assigned to functional and quality attribute (also known as non-functional) requirements compared to architectural considerations. As an example the Twin Peaks [24] model as adaptation of the spiral life-cycle model explicitly elevated architecture as a “peak” that together with the requirements “peak” is iterated upon.
early on in the software development process. Yet the possibility that architectural considerations 
can be as authoritative in nature as functional and quality attribute requirements is not 
sufficiently explored in theory and rarely applied in practice. Iterative, spiral, and agile processes 
remove the chronological linearity of lifecycle phases and tasks, but the authoritative flow of 
information and control remains largely top down (from requirements to architecture).

A third assumption among software architecture analysis and design methods and processes is 
that functional requirements, quality attribute requirements and design constraints are the driving 
forces that determine the selection of the appropriate software architecture for the system under 
development [5][23]. Specifically the major focus of current methods and processes is on 
explicitly identifiable and project related functional and quality attribute requirements.

Our research and industrial experience identify some limitations of the three assumptions listed 
above and we propose extensions to address those limitations to support practical applications.

The problem with the first assumption is the belief that all types of concerns can be elicited from 
the stakeholders and explicitly captured as requirements that can drive the selection of an 
appropriate architecture. We argue that there are more compelling contextual environment 
concerns that can be appropriately translated into macro-architectural heuristics, decisions and 
patterns through a decision analysis process relying on implicit knowledge, best practices and 
documented rationale.

In addition, we observe that certain architectural heuristics, decisions and patterns can be directly 
derived using implicit knowledge and best practices. Therefore, we seek to address the 
limitations of the second assumption and allow for requirements to be inferred from already
made architectural decisions and formally captured as explicit architectural requirements. We call such requirements backward-inferred macro-architectural requirements.

Finally, the third assumption does not fully account for the realities of software development practice in which contextual environment concerns frequently determine the appropriateness of software product decisions. We introduce an additional driving force in the form of forward-inferred macro-architectural requirements that are explicitly captured by analyzing contextual environment concerns.

The approach that we developed introduces the concept of macro-architectural requirements as an intermediate artifact facilitating the selection of an appropriate software architecture that fits the context. Macro-architectural requirements are a way to address the limitations of the second and third assumptions as described above. We integrate those macro-architectural requirements with the multilevel context-aware software architecture process to improve the software lifecycle efficiency and the quality of the target software product.
Figure 7 below builds on top of Figure 6 by showing how requirements are elicited and inferred. The figure shows an overview of forward-inferred and backward-inferred macro-architectural requirements within the context of the multilevel context-aware software architecture process:

The shaded boxes in the figure represent the macro-architectural requirements. There are two types of macro-architectural requirements – forward-inferred and backward-inferred – that differ by how they have been elicited.

![Diagram](image)

**Figure 7: Macro-Architectural Requirements.**

The forward-inferred macro-architectural requirements are inferred from the contextual environment–enterprise-level and system-level contextual environment concerns. Those explicitly captured, intermediate requirements serve both to influence and constrain the traditional software architecture process (described as micro-architecture), as well as to facilitate the decision analysis process that defines the macro-architecture (described in the form of macro-architectural decisions, heuristics, and patterns).
As shown in the figure above, backward-inferred macro-architectural requirements are inferred post factum from an already pre-defined macro-architecture. The macro-architecture that serves as the basis for the backward-inferred requirements is derived earlier from the contextual environment through decision analysis processes as described elsewhere [113]. Those explicitly captured, backward-inferred requirements serve to influence and constrain the traditional software architecture process described as micro-architecture in the figure above.

We propose a reverse requirements elicitation process through which we extract the backward-inferred requirements from the architectural decisions, heuristics and patterns which constitute the macro-architecture. The distinction between forward-inferred and backward-inferred macro-architectural requirements is somewhat arbitrary and depends largely on the practicality of capturing architectural requirements directly from the contextual environment (in the case of forward-inferred requirements) or from a pre-defined macro-architecture that serves as an architectural translation of some more elusive and implicit contextual environment concerns.

3.7.1 Forward Inferred Macro-Architectural Requirements

Our research and practical experience tell us that significant macro-architectural requirements inferred from the contextual environment are typically ignored and missed. We believe that the primary reason for this is the perception that architectural requirements are somehow secondary to the requirements elicited from business stakeholders. Evidently, some requirements with mostly technical and architectural implications are captured as either quality attribute requirements or design constraints. However, those are typically the obvious requirements that can be relatively easily detected by sophisticated business or senior systems stakeholders.
The more subtle architectural requirements typically can only be identified by experienced and knowledgeable architects. The more subtle requirements and concerns are usually delegated to the follow up software architecture and design processes. They are rarely captured as explicit requirements, and are considered as a lower priority with respect to functional and quality attribute requirements elicited directly from the business stakeholders. As a result, they are rarely considered with sufficient urgency while making the initial architectural decisions. Our experience tells us that serious issues can result from missing those macro-architectural considerations.

Forward inferred macro-architectural requirements are becoming especially meaningful as the software development and information technology industries are continuously evolving away from mostly “contextually greenfield” to predominantly “contextually constrained” development and operational environments. Contextual environment concerns that should now play a significant role in the selection of the appropriate software architecture include legacy libraries for reuse, legacy applications for integration, legacy and dominant development and operational skill sets within the organization, geographic location of development and operational units, maturity and track record of development and operational units with functional domain, technologies, and methodologies, strategic technology vendor partnerships and long lasting licensing agreements, among others.

The availability, desirability and inherent constraints of third-party libraries and commercial off-the-shelf (COTS) products can be appropriately modeled as forward-inferred macro-architectural requirements. (See related work [27].) Several recent trends have further increased the usefulness of macro-architectural requirements, including cloud computing and open source software.

Cloud computing in its different forms (Software as a Service / SaaS, Platform as a Service /
3.7.2 Backward Inferred Macro-Architectural Requirements

As we explain in Example 1 in section 3.6, some contextual environment concerns are hard or impractical to elicit as explicit requirements. Some of those more subtle contextual environment characteristics are most naturally translated into macro-architectural heuristics, decisions and patterns as a result of best practices, rules of thumb, and architectural experience. Once those contextual environment characteristics are implicitly translated in a target macro-architecture and documented as decision rationale and justification, some key architectural requirements can be reverse elicited and explicitly captured as backward-inferred macro-architectural requirements.

Looking at Example 1, the selected macro-architecture that translates the described contextual environment may include an explicit integration contract rigorously defined as a Web Services Description Language (WSDL) platform independent interface. The interface accommodates the geographically distributed development organization that has expertise and legacy functionality in several heterogeneous technologies. The geographical and technological distribution of expertise will result in a functional partitioning of architectural components that will be developed independently, modified frequently, and yet integrated seamlessly with little overhead.

Some details of the proposed architecture and approach are best captured implicitly as part of the justification and rationale of the architectural decisions. There are other specifics that are best-captured explicitly as macro-architectural requirements. It could have been hard and impractical
to try to elicit them in a forward manner from the contextual environment; however, they can be extracted quite easily from the defined macro-architecture and its rationale and justification.

Some trends in the software development and information technology industries make the backward-inferred architectural requirements especially applicable. The wide adoption of Service Oriented Architecture (SOA) as an architectural style applicable to numerous scenarios is one such major trend. Selecting the appropriate architectural style and respective technology standards can be implicitly captured as a justification for the macro-architectural decisions, but then some key architectural decisions should be explicitly extracted and documented as backward-inferred macro-architectural requirements. (See WS* web services standards and [17] for RESTful web services architectural style).

Another major trend is the coming of age of software ecosystems as a widely adopted model for software development and delivery [104]. In many current scenarios the development and delivery of software functionality relies on a software ecosystem to be economically viable and practical. Software ecosystems include the development and delivery of mobile application through the Apple iStore and Google Play (formerly known as the Android market), accepting online payments from payment ecosystems like PayPal. An additional example is given by healthcare applications compliant with the HL7 reference information model for semantic interoperability and with the IHE technical framework and integration profiles. These ecosystems entail some very involved and sophisticated architectural decisions, which would be hard to capture as forward-inferred macro-architectural requirements. They can be easily expressed as significant backward-inferred architectural requirements by a reverse elicitation process.
3.7.3 REVERSE REQUIREMENTS ELICITATION PROCESS

We proposed [40] a reverse requirements elicitation process as the mechanism through which backward-inferred macro-architectural requirements can be extracted from the macro-architectural heuristics, decisions, and patterns that constitute the macro-architecture of a system.

The reverse requirements elicitation process relies on architectural justification and rationale to be included in the documentation of the macro-architecture. Once the macro-architecture is defined and documented as an implicit translation of the enterprise-level and system-level contextual environment concerns, a cross-functional team consisting of a senior software architect and a senior requirements analyst analyze the macro-architectural decisions and heuristics and extract explicit requirements in the same form used to capture functional and quality attribute requirements from the different stakeholders. In Example 1, the following set of backward-inferred macro-architectural requirements can be extracted, assuming an application project from the insurance industry.

- The system shall be partitioned into subsystems that are loosely coupled and integrated through standard Web Services implementations and described in WSDL.
- The WSDL interface definitions will be centrally defined and managed by the shared service department in San Jose, California.
- The pricing engine shall be developed by the rating team in South Carolina as an extension to the existing Cobol pricing engine. The pricing engine will be accessible through the standardly defined WSDL interface as specified by the shared services department.
• The benefit contract engine shall be developed by the account benefits team in Virginia as an extension to the existing Java contracting engine. The benefit contract engine will be accessible through the standardly defined WSDL interface as specified by the shared services department.
4  PROBABILISTIC MACRO-ARCHITECTURAL DECISION FRAMEWORK

In the earlier sections we defined several important concepts and techniques to expand the software architecture practices to better handle the contextual environment of the system under development. In this section we will introduce a formal framework that implements the concepts and techniques in a consistent and systemic way by adapting the Probabilistic Graphical Models formalism and more specifically Bayesian Networks and Decision Networks a.k.a. Influence Diagrams. This section describes the Probabilistic Macro-Architectural Decision Framework (PMADF) based on the research published in [83].
4.1 MACRO-ARCHITECTURE AS DECISION ANALYSIS PROBLEM

We have advanced the proposition that the software architecture analysis and design process is a decision analysis discipline. We also introduced the contextual environment concept as an approach to support the design of an appropriate architecture for the system under construction that best fits the enterprise-level and system-level context [22][113].

In [113] we introduced the concept of macro-architecture – a set of broad, coarse grain, and highly impactful architectural decisions that can be made early on in the design process and that can be inferred and rationalized from the overall enterprise-wide and system-wide contextual environment. The concept of macro-architecture and macro-architectural requirements is best understood within the context of “decision frame” as presented in decision analysis theory.
Decision analysis introduces the decision hierarchy as a useful tool to define the frame for a decision problem. The theory and practice of decision analysis identify the framing of the decision focus as a critical step in the decision process. The middle band in the decision hierarchy is the decision frame - the focus of the decision process - where the strategic decisions should be made. In the top tier are policy decisions, an umbrella designation for assumptions and preexisting decisions that constrain the problem under consideration. Finally, the lowest tier covers tactical decisions that can be decided later without affecting the quality of the decision process.

As part of our proposal to approach the early phase of the architecture analysis and design as a decision analysis problem we have mapped the decision hierarchy to an architectural decision hierarchy model.
We capture explicitly and include the contextual environment as part of the architectural process.

We have observed that the contextual environment concerns—whether at the enterprise-level or the system-level—constrain the solution space to a subset of appropriate architectural options. We have observed that the contextual environment has not been traditionally treated explicitly. As a result the constrained solution space has been approached only implicitly. Our approach captures explicitly the constrained solution space that is affected by the contextual environment as a macro-architecture. The traditional software architecture—which we call micro-architecture—is constrained by the macro-architecture.

Figure 9: Architecture Decision Hierarchy Model.
4.2 PROBABILISTIC GRAPHICAL MODELS

We assessed several formalisms that would allow us to model and perform architectural decision making under uncertainty and selected probabilistic graphical models as the most promising.

Probabilistic graphical model is a formalism to encode knowledge about an uncertain domain. Probabilistic graphical models are a form of declarative representation that separates the knowledge from the inference on that knowledge. Knowledge is represented with a graph where nodes represent variables in the domain and the edges represent direct probabilistic relationship between the nodes. Probabilistic graphical models represent knowledge in a very compact way, since the structure of the graph encodes conditional independence among variables.

There are two major types of probabilistic graphical models – Markov networks based on undirected graphs – and Bayesian Networks – based on Directed Acyclic Graphs (DAGs).

Markov networks have been successfully applied to model lower level activities in image processing and natural language processing.

Bayesian networks have been extensively applied in Artificial Intelligence and widely used to model inference and learning under uncertainty. There are efficient inference algorithms on Bayesian networks that make probabilistic inference practical in real-world scenarios. The Decision Network or Causal Bayesian Network variant of Bayesian networks provide powerful extensions that allow us to perform efficient decision making under uncertainty. We adopted Bayesian Networks and more specifically Decision Networks to model macro- architectural knowledge and perform probabilistic architectural decision making.
4.2.1 BAYESIAN NETWORKS AND DECISION NETWORKS

PMADF is a concrete implementation of these concepts, adapting the formalism of BN and more specifically the decision networks variant that includes enhancements to support causality, interventions and decision making [83].

Bayesian networks are probabilistic graphical models that capture the uncertainty and conditional dependencies of the world. Probabilistic graphical models integrate graph theory and probability theory to capture knowledge in an uncertainty domain. Bayesian networks have a graphical structure (“qualitative part”) with nodes and edges where nodes represent model variables and edges indicate conditional dependencies between related variables. The graphical structure of the Bayesian network captures the conditional dependency of variables. By expressing conditional independence between two variables through a missing edge connecting the nodes, Bayesian networks greatly reduce the computational complexity of the model. Conditional probabilities (“quantitative part”) are captured in the form of Conditional Probability Tables (CPT) at each node that reflect the conditional dependency of the node on its parent nodes. Initial CPT probability values reflect the prior probabilities identified either through expert level opinion or historical data. Computationally efficient algorithms exist to traverse graphs that make Bayesian networks a very practical formalism for knowledge representation and reasoning.

Decision networks (aka Influence Diagrams) introduce decision nodes and utility nodes in addition to the chance nodes that are part of Bayesian networks. While Bayesian networks are usually interpreted to reflect conditional probabilities and not necessarily a causal relationship between variables, decision networks carry the semantics of a causality relationship between nodes connected with a directional edge [126].
We chose the decision network formalism to implement PMADF because of its capability to capture uncertainty and conditional dependency and its ability to handle prior probability distributions identified from subject matter expert opinion and historical data. A decision network can accommodate the consistency of the model even as new potentially conflicting evidence is entered. A decision network can also express causal relationship and decision making by incorporating the concept of utilities to compare the desirability of different solutions.

For the specific technology implementation we chose Netica from Norsys as a feature-rich and proven implementation of Bayesian Networks and Decision networks. [127]
4.3 MODELING CONTEXTUAL FACTORS, PREFERENCES AND CAUSAL RELATIONSHIPS

The macro-architectural decision framework adapts the BN formalism and more specifically its extension—decision networks. The decision framework consists of the following architecturally relevant components: contextual factors, macro-architectural preferences, macro-architectural metrics, and macro-architectural profiles.

The components of the decision framework, their parameters and their relationships expressed by the structure of the probabilistic graphical model capture the essential semantics, causality, uncertainty and utility of the architectural scenario under consideration. What follows is a more detailed description of each component in the framework.

Contextual Factors (CF) are an enhancement to the Decision Pyramid framework. They are divided into two categories: contextual constraints (CF_CNSTR) and contextual considerations (CF_CNSDR). Contextual constraints are factors taken for given—these are externally controlled factors that are considered non-negotiable with a very high probability that they will not change. For the purpose of the decision framework we consider the probability that they will not change to be 1.

Example: The software our organization develops controls mission-critical operational processes of a nuclear power station and it can be taken as a given that every software module will be tested with our certified internal testing team.

Contextual considerations are external factors and generally outside the architect’s immediate influence, however they are not deterministically fixed. The probability that they will not change is less than 1 and the probability of them staying as projected or changing is a value that can be predicted by the architect based on expert level knowledge.
Contextual considerations are further categorized as objectives, projections of future value, and options. However, this additional categorization is for clarity and does not have a material impact on the probabilistic nature of the considerations as defined in the decision framework.

- Objectives are contextual factors that have been specified as goals by the development or sponsoring organization and are actively being managed to be achieved. They have some probability to be achieved.
  
  - Example: *The goal of our company is to achieve 20% market share. There will be zero critical security vulnerabilities in our software system.*

- Projections are contextual factors that have a quantified value in the future. That value projection may end up being correct or wrong and a probability can be assigned to that value being correct.
  
  - Example: *By 2018 we project that 60% of all software across industry will be accessed from mobile devices.*

- Options are contextual factors that are presented as alternatives, and the architect has little freedom to choose among the alternatives (if any). Although options may have some similarity with requirements they are contextual in nature so it is more appropriate to model them as probabilistic alternatives.
  
  - Example: *The new system will be developed initially on either Linux or Windows.*
    
    *(The choice is not random, but probabilistic.)*

Macro-Architectural Preferences (MAPs) are strategic, highly impactful decisions that frequently are viewed as strategic/business, organizational, financial, operational, or operational decisions.
In practice some of these decisions have a significant direct or indirect impact on the architecture of the software system under development and the likelihood of the selected architecture to be successful. Those decisions are defined as macro-architectural decisions and the resulting architectural model specifies the macro-architecture of the system. In the macro-architectural decision framework we named them preferences to emphasize their probabilistic nature. They are decisions made at the macro-architectural level that serve as preferences in the following micro-architectural and design phases.

Macro-Architectural Metrics (MTRC) measure the macro-architecture to assess the desirability of macro-architectural decisions and compare decision alternatives. The macro-architectural metrics are divided in two categories: objective measures (MTRC_MSK) and subjective utilities (MTRC_ULT).

Objective measures are directly measurable quantities such as cost (e.g., development cost or total cost of ownership), time (e.g., time to build, time to market), and quality (e.g., number of defects, number of patches). Subjective utilities measure the perceived value or desirability of a macro-architecture. They are typically measured indirectly through calibrated survey and questionnaire techniques from stakeholders and address more abstract concepts such as stakeholder satisfaction, client satisfaction, system success, and so on.

Some metrics can be viewed as either objective measures or subjective utilities depending on how they are collected and measured. Some examples include complexity, flexibility, evolvability, and risk [130]. Where possible we prefer to treat them as objective measures. Complexity can be measured by the number of architectural building blocks—components interfaces, communications, persistence, user interaction. Flexibility can be measured as time
(e.g., hours, days or months) to add new features in the same usage domain. Evolvability can be measured as time (hours, days or months) to add new features in order to support a new usage domain. Risk can be measured by probability to deliver on time, on budget, and with expected quality. However, since predicted probability is a subjective characteristic, it may be considered a utility.

Macro-Architectural Profiles (PRFL) group common patterns of contextual factors, optimal macro-architectural decisions and their expected metrics. They help us quickly instantiate the decision framework for a recurring environment. Examples include a start-up company profile, utility company profile, and so on.
5 CASE STUDIES AND RESULTS

We have applied PMADF to different cases from software architecture development practice. In this section we describe two case studies and report the results.
5.1 SERVICE ORIENTED ARCHITECTURE DECISION

To demonstrate how the macro-architectural decision framework works we will walk through a case study of its application for deciding if Service Oriented Architecture (SOA) should be the primary architectural style for a system implementation at a healthcare company. The case study demonstrates how the decision framework is used to inform and automate architectural decisions reproducing the outcomes of the manual architecture design process. In addition to providing evidence of the feasibility of the decision framework approach, the case study demonstrates how the probabilistic BN supports exploration of what-if scenarios and updates to information as new evidence emerges during system design.

John Doe was the software architect assigned to lead the design and implementation of the system under study—a survey and clinical performance assessment system for self-reported clinical performance data from contracted medical facilities that would be used to assess the quality of provided healthcare services and adherence to clinical protocols. Like many contemporary software implementations the system would most likely integrate other pre-existing or commercial systems as a system of systems and SOA would likely be a candidate for the primary architectural style.

John developed a probabilistic decision model using the decision framework that allowed him to identify appropriate architectural preferences early on based on relevant contextual factors. One architectural decision John had to make was whether to use SOA as the primary architectural style. In the decision framework this decision is modeled as the macro-architectural preference (MAP_01), since it is probabilistic in nature and serves as a recommendation and directional input to later design phases of the design process. So the preference under consideration was whether a Service Oriented Architecture (SOA) should be used.
At the early stages of product development John analyzed several relevant contextual factors. Three contextual considerations that could make SOA an appropriate architectural approach were the future reuse of the system (CF_CNSDR_01), the development method that the organization would use to develop the system (CF_CNSDR_02), and the availability of SOA enabled subsystems developed by 3rd party vendors for part of the functionality (CF_CNSDR_03). System reuse was an objective of the company, and John assessed that there was a certain probability that the objective would be achieved. The development method was a pre-selected option, and John assessed that there was a certain probability that the option would hold as it depended on the knowledge and experience of the development team with the development methods. The availability of SOA enabled subsystems from 3rd party vendors within the next couple of years was a projection that John assessed to be quite uncertain as well.

John knew that the relevant architectural measures to using an SOA approach were the cost to develop an SOA system and the inherent complexity of an SOA approach. These two metrics of architectural quality could be quantified as objective measures and minimized. Since decision networks maximize the values of the utility nodes, affordability (MTRC_MSR_01) and simplicity (MTRC_MSR_02) were used as the metrics instead of the more intuitive opposites cost and complexity which would have to minimized.

Additionally the subjective utility customer satisfaction (MTRC_UTL_01) provided a comprehensive metric to quantify the desirability of the architectural alternatives. John used techniques from utility theory to measure the subjective composite metric that captured customer perception for implementation risk and final system flexibility.
John also analyzed other relevant contextual factors and identified three contextual constraints that could safely be considered as given—the team that would be responsible to develop the solution and its experience with development methodologies and related skillset (CF_CNSTR_01), the source of funding for the project (CF_CNSTR_02), and the positioning of the system as either a strategic enterprise asset, function point project or operational efficiency project (CF_CNSTR_03).

In summary, John identified the following components of the macro-architectural decision model:

- CF_CNSTR_01 – Development Experience and Skillset
- CF_CNSTR_02 – Project Funding Source
- CF_CNSTR_03 – Product Positioning
- CF_CNSDR_01 – System Reuse
- CF_CNSDR_02 – Software Development Method
- CF_CNSDR_03 – 3rd party SOA enabled subsystems
- MAP_01 – Use Service Oriented Architecture (SOA)
- MAP_02 – Use 3rd party SOA enabled subsystem
- MTRC_MSR_01 – Affordability
- MTRC_MSR_02 – Simplicity
- MTRC_UTL_01 – Customer Satisfaction
The figure below shows the screenshot that captures the probabilistic decision model as implemented in Netica:

![Decision Network Implementation](image)

**Figure 10: Decision Network Implementation.**

The top three chance nodes capture the relevant contextual factors that are constraints; the three chance nodes below them are contextual factors that are considerations. On the third level are
decision nodes that capture the macro-architectural preferences that are inferred from the model. On the lowest level are the utility nodes that capture the objective measures and subjective utilities for the model. The network graph structure captures the conditional dependency and causality. A separate CPT captures the conditional probability distributions.

From the model structure we can observe that there is conditional dependency between the development team skills and experience constraint (CF_CNSTR_01) and the system reuse consideration (CF_CNSDR_01). This can be viewed from a pure probabilistic perspective as conditional dependency between the two variables (basic BN model) or as a causal relationship from a causality perspective (extended decision network model). The conditional dependency can be inferred from historical data, as was done in this study. Additionally, there is a causal relationship that was identified through knowledge extraction from domain experts. The causal relationship was confirmed by the historical project data; intuitively that causal relationship means that the level of reuse of a system depends on the skill and experience of the development team.

If the development team is skilled and experienced in building external client-facing software products that are deployed multiple times, then it is more likely that the resulting software system will end up being reused more than once or twice. If the development team is primarily experienced developing internally-focused IT systems for operational efficiency, then it is less likely that the resulting system will be reused more than two or three times.

If the development team is an external consulting company the causal relationship is still existent, but more subtle. External consulting companies are more likely to focus on developing software point systems with clear requirements and timelines that allow predictable financial
closure and that would favor less system reuse; however, they also have a deeper skillset and more varied experience than predominantly internal IT teams and that results in higher likelihood for system reuse. The causal relationship is in accordance with the historical data and experience of the company in this study.

The reason that the system reuse node distinguishes one time system reuse, two times system reuse, and three or more times system reuse is based on the economics of software and service reuse, where the financial break-even point and positive return on investment (ROI) has been shown to typically occur at between two and three times of system reuse [131]. The break-even analysis and ROI calculations used were based on the economics of software reuse whereby using family of products and software product lines as proxies for SOA economics accurately simplified the modeling problem.

There is also a conditional probability and causal relationship between the funding source constraint (CF_CNSTR_02) and the system reuse consideration (CF_CNSDR_01). The intuition behind this causal relationship is straightforward and is again validated by historical project data—projects funded from product marketing as profit centers are more likely to be reused three times or more (as they are developed as products to be sold to many profitable clients), while projects funded from cost centers at the departmental level are much more likely to be point solutions with little opportunity to be reused. Projects funded as enterprise investments have a higher likelihood to be reused at least two times, but not to the same extent as profit center client products.
The following CPT shows the prior probabilities for the system reuse consideration node (CF_CNSDR_01) derived from historical project data and expert opinion solicited from architects and business stakeholders at the case study company. It is important to point out that the table reflects the conditional probability from all parent constraint nodes that influence the system reuse consideration node—they happen to be CF_CNSTR_01, CF_CNSTR_02, and CF_CNSTR_03.

As can be seen from the decision network structure there is a conditional probability and causal relationship between the development team skills and experience constraint (CF_CNSTR_01) and the development method consideration (CF_CNSDR_02). The intuition behind this causal relationship is again validated by historical project data—teams with skill and experience developing external client-facing software products are more likely to use a software product development methodology, while teams primarily experienced with developing internally-
focused IT systems for operational efficiency are more likely to use either an internal multiuser system development methodology or a utility tool development methodology with comparable frequency.

External consultant development teams tend to favor an internal multiuser development methodology, but can step up to the more expensive product methodologies when those consulting opportunities arise. The development method consideration node (CF_CNSDR_02) has conditional dependency on the constraint nodes CF_CNSTR_01, CF_CNSTR_02, and CF_CNSTR_03 as can be observed from the decision diagram. The CPT for the development method consideration node is given in the screenshot below:

Figure 12: CPT for Development Method.
The following CPT captures the conditional dependency for the availability of 3rd party SOA clients for the system (CF_CNSDR_03) which is influenced by the funding source constraints (CF_CNSTR_2).

![CPT for SOA Clients Available](image)

**Figure 13: CPT for SOA Clients Available.**

The next few screenshots capture the value assignments for the architectural metrics. For the SOA architectural decision model they are affordability, simplicity, and customer satisfaction from the implementation.

For affordability (MTRC_MSR_01) the values are:

![Utility Table for Affordability](image)

**Figure 14: Utility Table for Affordability.**
For simplicity (MTRC_MSR_02) the values are:

![Utility Table for Simplicity](image)

**Figure 15: Utility Table for Simplicity.**

For customer satisfaction from the implementation (MTRC_UTL_03) the values are:
Once the macro-architectural decision model is captured in the decision network, the network is compiled and ready for queries and inferences.

Without explicitly setting the prior probability values for the constraints the BN is initializing those values to uniform probability distribution (e.g., one third for each possible value). If that were the case, we can observe that the Macro Architectural Preference “Use SOA” (MAP_01) is inferred to be “Build Not SOA” with a value 230 (the expected utility for “Build SOA” is only 118). Such a scenario is not valid since by definition the contextual constraints are not probabilistic in nature, but rather deterministically preset within the context of the development project.

Figure 16: Utility Table for Customer Satisfaction.

Once the macro-architectural decision model is captured in the decision network, the network is compiled and ready for queries and inferences.
5.1.1 SCENARIO ANALYSIS

To demonstrate how the macro-architectural preferences are different depending on the contextual factors we experiment with several scenarios.

Scenario 1:

The initial model consists of the following factors

- Development team is experienced with product development methods for external products
- Product is funded in a profit center model from marketing
• Product is strategic for the company

The resulting Macro-Architectural Preference is “Build SOA” (266 for “Build SOA,” 162 for “Build Not SOA”).

![Decision Network Diagram](Image)

**Figure 18: Scenario 1 Decision Network.**

Scenario 2:

Changing the funding source from profit center to enterprise investment doesn’t change the resulting preference

• Development team is experienced with product development methods for external products

• Product is funded as enterprise investment

• Product is strategic for the company
The Macro-Architectural Preference is “Build SOA” (235 for “Build SOA,” 159 for “Build Not SOA”).

Scenario 3:

Switching the product positioning from strategic to functional point project and from external product experienced team to internal IT experienced team changes the preference to Build Not SOA

- Development team is experienced with internal IT method
- Product is funded as enterprise investment
- Product is a functional point project


![Decision Network Diagram]

Figure 20: Scenario 3 Decision Network.

Scenario 4:

Having a team with external products experience in place of the team with primarily internal IT experience weakens the preference for “Build Not SOA,” but does not change the preference.

- Development team is experienced with product development methods for external products
- Product is funded as enterprise investment
- Product is a functional point project

The Macro-Architectural Preference is “Build Not SOA” (166 for “Build SOA,” 202 for “Build Not SOA”).

Figure 21: Scenario 4 Decision Network.

A powerful feature of the decision framework is the ability to update the probability values as new evidence becomes available, thus dynamically adjusting the recommended preferences with the new information.
For example, in this scenario the recommended preference to “Build Not SOA” is based on expected conditional probabilities extracted from historical project data and initial subject matter experience. If additional information is made available, such as there is a higher likelihood that the system will be reused three or more times (e.g., because a couple of additional projects are committing to reuse the system) then the Bayesian network recalculates the preference recommendations using the updated conditional probabilities, while still maintaining consistency with the other prior probabilities. We can see that the decision network calculates that the conditional probability for three or more reused systems is calculated at around 34%.

Scenario 5:

If there is evidence that reuse is much more likely, we can insert the new information as likelihood and revise the model. Based on new information we inserted a likelihood of 75%.

The Bayesian network recalculates the conditional probabilities incorporating this new information. As shown in Figure 23 for the updated Scenario 5 the probability for a reuse of three times or more increases to 79%. It is important to note that the new information was

Figure 22: New System Reuse Likelihood Evidence.
entered as likelihood—it still comes from an uncertain source and there is a probability that new evidence may not be true. We could have inserted the new information as fact, which would have adjusted the network to have a 100% probability (if for example there is a guarantee that the system will be reused); however, that would not have been a factually-accurate assumption in the study being described.

Inserting the newly acquired information that the likelihood for reuse of three times or more is 75% (two times set at 20% and one time set to 5%) changes the preference as shown in the figure below.

- Development team is experienced with product development methods for external products
- Product is funded as enterprise investment
- Product is a functional point project

The macro-architectural preference now is “Build SOA” (233 for “Build SOA,” 153 for “Build
In the case study Scenario 4 captures the initial model that was developed. The company already had previous experience with both successful and unsuccessful SOA system development and was eager to make the proper decision. As part of the architectural decision process Scenario 5 was developed and analyzed and that architectural preference to build the system using SOA was selected.
5.2 “OPEN SOURCE OR COTS” AND “DSL OR CONFIG FILE” DECISIONS

In [113] we reported on a case study that we performed at a US-based company during the development of three versions of a major software system. The software architecture of the system had evolved over three major releases and we captured the key architectural decisions made during those releases. We analyzed the impact of introducing contextual factors and macro-architectural preferences in the architectural design process and reported on the successful outcomes of applying the Probabilistic Macro-Architectural Decision Framework. The following two macro-architectural preferences and subsequent architectural decisions were part of the probabilistic macro-architectural decision model – in this section we show the Norsys Netica implementation of the model:

- **Open Source vs. COTS**: should part of the system functionality be fulfilled using an open source library or Commercial Off The Shelf (COTS) package
- **Configuration File vs. DSL**: Should the system configurability be achieved through a configuration file or through a custom developed Domain Specific Language (DSL)

The following screenshot from Norsys Netica shows the probabilistic macro-architectural decision model that allows us to reason about these architectural decisions and perform scenario analysis to select the optimal software architecture.
The following architectural constraints and considerations were identified:

- **CF_CNSTR_01 – Development Sourcing**
  - Team Developing External Customer Software Products
  - Team Developing Internal IT Systems
  - Outsourced to Consulting Team

- **CF_CNSTR_02 – Funding Source**
  - Department Cost Center
  - Enterprise Investment
  - Profit Center from Product Marketing

- **CF_CNSTR_03 – Cost Culture**
  - Focus on Lowest Cost Possible
  - Focus on Reliable Cost Estimate

- **CF_CNSTR_04 – Business Team Technology Profile**
  - Technically Savvy Hands-On Business Team
  - Technically Uninvolved Hands-Off Business Team
- CF_CNSDR_01 – Development Team Experience
  - New Large Scale Development
  - Maintenance of Existing Software
  - System Integration and Configuration of Vendor Packages
- CF_CNSDR_02 – Software Maintenance Model
  - Internal Software Maintenance Team
  - Outsourced Software Maintenance Team
- CF_CNSDR_03 – Support Staff Focus
  - Hands On Developers Supporting Software
  - Technicians Passing Through Issue Reports to External Vendors
- CF_CNSDR_04 – Sponsoring Company Quality Profile
  - Expects No Defects At Launch
  - Tolerated Defects At Launch (Expecting To Be Quickly Fixed)

The following Macro-Architectural Preferences (MAPs) were calculated as the target of the model:

- MAP_01 – Use “Open Source” or “COTS”
- MAP_02 – Use “Configuration File” or “Domain Specific Language (DSL)”

And finally, the following macro-architectural metrics were selected to quantify the objective value and subjective utility of the different options:

- MTRC_MSR_01 – Affordability
- MTRC_MSR_02 – Certainty
- MTRC_UTL_01 – Solution Flexibility
As described in [113], two versions of the software system were considered failed implementations even though they were based on competent software architectures in a technical “fit for purpose” sense. The architectural problems that were uncovered were related to contextual factors and inappropriate architectural decisions from a macro-architectural “fit to context” perspective.

As an example, the development was outsourced to a highly technically qualified external team, while the software maintenance was assigned to an internal team that was skilled in passing through issue reports to external vendors and not internal hands-on fixes. The contextual environment also included a company culture that put a premium on meeting cost estimate commitments and not on achieving the lowest development cost possible. In addition, the stakeholder metrics revealed strong preference for certainty (risk averse) and low to moderate focus on affordability. When the contextual factors for this use case were entered in the decision model it became evident that the otherwise technically solid software architecture based on an open source library was inappropriate for this contextual environment and that a Commercial Off The Shelf (COTS)-based architecture was a much better choice.

The “Configuration File” vs. “Domain Specific Language (DSL)” architectural decision was a harder choice, but again the model based on the Probabilistic Macro-Architectural Decision Framework provided a correct guidance. The second decision was not as clear cut, since the stakeholders were putting a very high subjective utility on “solution flexibility” – they wanted to configure or almost customize the solution to serve very different needs which favored using a powerful DSL. However, a contextual constraint where the business users were rather
uninvolved and hands off with the technology and thus unlikely to comfortably use the DSL paired with a contextual consideration that the organization would not tolerate an expected lower initial quality of a more complex DSL technology skewed the macro-architectural preference in the direction of the less flexible, but simpler to use and develop configuration file approach.

Experimenting with different likelihoods of the contextual considerations allowed the software architect to eventually converge on the correct architectural choice with version 3 of the system as reported in the paper.
6 CONCLUSIONS AND FUTURE RESEARCH

As part of our academic research and industry experience we identified an important problem within the software architecture community of practice – technically solid software architecture decisions sometimes resulted in contextually inappropriate software architectures and as a consequence in failed software system implementations. We were rather intrigued by the problem and perplexed by the apparent inability of existing methods and techniques to solve it. We observed that the mainstream approaches in the software architecture community to improve the requirements elicitation methods and to expand on the traditional structural decomposition design techniques were not yielding substantial improvements to the outcomes. We recognized that a novel approach was needed.

A few major insights and contributions that we made included:

- Recognizing that not all forces that impact the software architecture are explicitly stated and can be captured as requirements or quality attributes
- Recognizing that some strategic / business, operational, financial, organizational, and cultural forces that typically are not considered during software architecture development should also be taken into consideration
- Formally defining and modeling those new forces as contextual factors
- Introducing a multi-level software architecture design process where a macro-level models and addresses contextual factors and related architectural preferences
• Recognizing that the macro-architectural level is better approached as a probabilistic
decision analysis problem, while the remaining architectural design can continue to be
approached with structural decomposition engineering methods

• Developing a Probabilistic Macro-Architectural Decision Framework (PMADF) based on
an adaptation of Bayesian Networks (BN) and Decision Network formalisms

• Performing empirical case studies to test out and validate the probabilistic macro-
architectural decision framework

• Providing categorization of contextual factors and macro-architectural preferences to
enable software architecture practitioners to apply the decision framework in their work

We argue that the framework and techniques that we introduced have a significant impact on the
software architecture theory and practice and is well positioned to improve the quality of
software architecture practices and the software development outcomes.

We are passionate about our work and intend to continue to develop and improve the methods
and supporting tools. We intend on continuing our research in this area and suggest the following
research problems as extensions to our current work. We suggest extending the catalogue of
contextual factors, macro-architectural preferences and causal dependencies by automating the
process of discovery of new factors, preferences and their relationship through data mining and
machine learning techniques. One of the powerful features of Bayesian Networks is that they can
be easily used to learn the network parameters and also the network structure from historical data
using machine learning algorithms. This is a very powerful approach and promising area of
extending our research and is on the short list of our future research activities.
REFERENCES


[51] C. Hofmeister, P. Kruchten, R. L. Nord, H. Obbink, A. Ran, and P. America,


267–270.
NAME: Plamen P. Petrov

EDUCATION:
B.S., Computer Science, Technical University of Sofia, Sofia, Bulgaria, 1989
M.S., Computer Science, Loyola University Chicago, Chicago, Illinois, 1992
Ph.D., Computer Science, University of Illinois at Chicago, Chicago, Illinois, 2014

TEACHING:
College of Computing and Digital Media, DePaul University, Chicago:
Service Oriented Architecture, 2011-2013
Department of Computer Science, Loyola University Chicago, Chicago:
Enterprise Software Architecture, 2007-2012
Software Quality and Testing, 2007-2011
Advanced Operating Systems, 1999-2002

PROFESSIONAL MEMBERSHIP:
Association for Computing Machinery

PATENTS:
Alarm manager system for distributed network management system
United States 6356282
Michael Roytman, Plamen Petrov
Issued March 12, 2002

Distributed computer platform with flexible configuration
United States 20040158575
Christian Jacquemot, Frederic Herrmann, Plamen Petrov, Francis Prusker
Filed August 12, 2004

PUBLICATIONS:
