A Framework for the Characterization and Analysis of Software Systems Scalability

Ana Leticia de Cerqueira Leite Duboc

Thesis submitted for the degree of

Doctor of Philosophy

of

UCL

Department of Computer Science

University College London

2009

Abstract

The term *scalability* appears frequently in computing literature, but it is a term that is poorly defined and poorly understood. It is an important attribute of computer systems that is frequently asserted but rarely validated in any meaningful, systematic way. The lack of a consistent, uniform and systematic treatment of scalability makes it difficult to identify and avoid scalability problems, clearly and objectively describe the scalability of software systems, evaluate claims of scalability, and compare claims from different sources.

This thesis provides a definition of scalability and describes a systematic framework for the characterization and analysis of software systems scalability. The framework is comprised of a goal-oriented approach for describing, modeling and reasoning about scalability requirements, and an analysis technique that captures the dependency relationships that underlie typical notions of scalability. The framework is validated against a real-world data analysis system and is used to recast a number of examples taken from the computing literature and from industry in order to demonstrate its use across different application domains and system designs.

Acknowledgements

"By the end of it, you will hate scalability with all your strength!" warned me a fellow PhD student. Perhaps strangely, I enjoyed my research to the last day and I am still fascinated by the subject of scalability. This would never be possible without the support of some great people:

David Rosenblum and Tony Wicks, who became role models for me. I hope I have managed to learn at least a few of their many qualities as outstanding computer scientists, researchers and supervisors. Thank you for all these years of supervision and friendship!

Emmanuel Letier, who guided me to the fascinating world of requirements engineering. Thank you for the countless discussions, the words of encouragements, and all the knowledge you brought to this research.

Wolfgang Emmerich and Anthony Finkelstein, my examiners on the first, second and mock vivas. Your experience, knowledge and feedback have been invaluable to this research. Thank you.

Soren-Aksel Sorensen, Antonio Carzaniga, Sebastian Ebaum, Alex Wolf, Damon Wischik, Elaine Weiuker, and Thomas Alspaugh, world-class researchers with whom I had the opportunity to discuss this research.

Fortent staff, especially John Taylor, Andrew Towers, Steve Brewin, Iain MacLaren, Alexon Bell, Colin Briggs and Ben Cave, who gave me all the support I needed to complete the case studies. John deserves a especial thanks for over 10 years of help since I first joined Searchspace!

Capacitas staff, especially Andy Bolton, Manzoor Mohammed, Danny Quilton, and Paul Seaton-Smith, from whom I have learned a lot about performance and capacity planning.

My friends Vanessa Almeida, Carina Alves, Rami Bahsoon, Elias Bareinboim, Michael Browarski, Marcus Bulach, Hina Keval, Licia Capra, Jose Heleno Faro, Gorka Guardiola, Rae Harbird, Richard Mapes, Siva Murugiah, Christian Netwitch, Alok Rana, David Rawlinson, Genaina Rodrigues, Sonia Shaw, Helen Shum, Luis Silva, Socrates Varakliotis, Stefanos Zachariadis, with whom I had countless discussions on scalability and a great time during this PhD research. A special thanks for Franco, who were very helpful as my mock examiner.

My Brazilian and my English family: Maria Lucia, Gilson, Ana Luisa, Maria Sylvia, Isoel, Gizela and Pedro, who despite knowing nothing about scalability, gave me all the support that made this work possible. I cannot thank you enough.

Contents

List of Acronyms

ADL	Architecture Description Language
AHP	Analytic Hierarchy Process
ATAM	Architecture Tradeoff Analysis Method
CBAM	Cost Benefit Analysis Method
EJB	Enterprise JavaBeans
EPA	Experimental Program Analysis
ERP	Enterprise Resource Planning
ETL	Extraction, Transformation and Load
IEF	Intelligent Enterprise Framework
J2EE	Java 2 Enterprise Edition
JVM	Java Virtual Machine
LAS	London Ambulance Service
MDA	Model Driven Architecture
MMS	Multimedia Message Service
MTTF	Mean Time To Failure
OLAP	Online Analytical Process
OLTP	Online Transaction Process
OWL-S	Ontology Web Language for Services
PASA	Performance Assessment of Software Architectures
PNL	Performance Non-Scalability Likelihood
QoS	Quality of Service
QSEM	Quantitative Scalability Evaluation Method
RFP	Request For Proposals
RUP	Rational Unified Process

SAAM	Scenario-based Architecture Analysis Method
SAP	Session Announcement Protocol
SBAR	Scenario-Based Architecture Reengineering
SKServer	Surrogate Key Server
SLA	Service Level Agreements
SMS	Short Message Service
SPE	Software Performance Engineering
SQL	Structured Query Language
STAS	Scalability Testing and Analysis System
UAT	User Acceptance Tests
UML	Unified Modeling Language
WAP	Wireless Application Protocol
WSOL	Web Service Offering Language

List of Figures

List of Tables

Chapter 1

What Is Scalability?

The computing community is lacking (1) a precise definition of the term scalability and (2) a systematic, uniform and consistent treatment of scalability that can be applied across application domains and system designs. In this work, we (1) provide such a definition and (2) develop systematic techniques to characterize, analyze and compare the scalability of software systems.

1.1 Motivation

Since the origins of software development, the size and complexity of software systems have been ever increasing. Scalability is a critical quality for software systems and, as suggested by Figure 1.1, it is a quality that has been receiving growing interest over the years. But what exactly is scalability?

1.1.1 A Frequently Claimed Attribute

> *"I examined aspects of scalability, but did not find a useful, rigorous definition of it. Without such a definition, I assert that calling a system 'scalable' is about as useful as calling it 'modern'. I encourage the technical community to either rigorously define scalability or stop using it to describe systems."* —Mark D. Hill, *What is Scalability?* (Hill, 1990)

As this quotation suggests, *scalability* is a term that is poorly defined and poorly understood. It is an important attribute of computer systems that is frequently asserted but rarely validated in any meaningful, systematic way. Many computer scientists employ phrases such as "X is a scalable system" or "Y is not a scalable approach" in order to convey some intuition about the system or approach at hand, but such phrases leave little more than the vague and ill-formed impression that "X is good" or "Y is bad".

Although Hill's quotation is some 18 years old, it seems as valid today as when he first stated it. The term *scalability* continues to be widely used without precision in technical literature, including design documents, research papers, standards specifications and product brochures. As an illustration, consider the following quote extracted from a content management company's website:

> *"Scalability is a key requirement for the corporate content infrastructure, . . . [which] needs to be capable of handling high volumes of content as well as of fulfilling high performance requirements."* (EMC^2, 2008).

The company, named EMC^2, develops content management and distribution products and services for enterprise-level systems. This quote makes a fairly typical use of the term, in which scalability is used

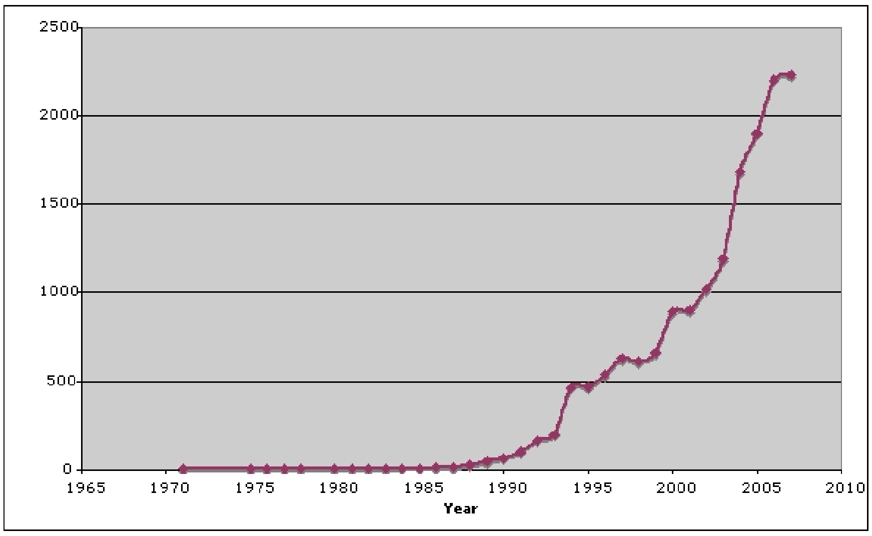


Figure 1.1: Publications with 'scalable' or 'scalability' in the title (source: Engineering Village 2).

to convey an intuition of high performance or high capacity. Typically, what is meant by scalable is not precisely defined or further refined, making it difficult to judge the veracity of the claim.

This is a typical example of semi-technical literature, marketing brochures and technical summaries, where precision is deliberately sacrificed in favor of conciseness and/or effect. However, the imprecise use of the term can also be found in more solid technical literature, like the specification of the Session Announcement Protocol (SAP), a well known protocol studied in networking (Handley et al., 2000). The roughly 5500-word document contains exactly three occurrences of the word *scalability*. The first occurrence is in the abstract of the document; here is the complete abstract:

> *This document describes version 2 of the multicast session directory announcement protocol, SAP, and the related issues affecting security and scalability that should be taken into account by implementors.* (Handley et al., 2000)

The second occurrence is in a paragraph in the middle of the document; here is the complete paragraph:

> *A SAP announcer periodically multicasts an announcement packet to a well known multicast address and port. The announcement is multicast with the same scope as the session it is announcing, ensuring that the recipients of the announcement are within the scope of the session the announcement describes (bandwidth and other such constraints permitting). This is also important for the scalability of the protocol, as it keeps local session announcements local.* (Handley et al., 2000)

The third occurrence is within a heading entitled “Scalability and Caching” for a section containing a free-form discussion of desiderata that neither uses the term nor precisely defines its intended meaning.

The authors of this document are extremely talented network protocol designers, and thus their claims about scalability arguably can be taken on trust. But what exactly are their claims? What is it whose scalability is being claimed? Is it the whole protocol? The caching strategy of the protocol? What system property is being measured to determine scalability? Is it message delay? Message throughput? What is it whose scaling is considered important? Is it something in the design of the protocol? Something in the execution environment of the protocol?

As in these examples, numerous are the cases in the computing literature in which scalability is mentioned in passing in this fashion or simply claimed outright but never fully defined or justified. The little consensus on the meaning of scalability gives rise to a range of competing, and at times inconsistent views on what it means for a system to be scalable. These different views may lead to misinterpretations of scalability claims and difficulty in evaluating claims or comparing statements of scalability from different sources.

1.1.2 A Poorly Defined Term

To overcome this problem, a few authors have attempted to define scalability, mostly in the form of rules of thumb, with varying degrees of rigor:

> *"Scalability means not just the ability to operate, but to operate efficiently and with adequate quality of service, over the given range of configurations."* (Jogalekar and Woodside, 2000)

> *"Load scalability: ability to function gracefully (i.e., without undue delay and without unproductive resource consumption or resource contention at light, moderate, or heavy loads) while making a good use of available resources."* (Bondi, 2000)

> *"A system is a scalable system if it can be deployed effectively and economically over a range of different 'sizes', suitably defined."* (Jogalekar and Woodside, 1998)

> *"Scalability is the ability to handle increased workload by repeatedly applying a cost-effective strategy for extending a system's capacity."* (Weinstock and Goodenough, 2006)

Terms such as "efficient", "effective" and "adequate" are commonly found in definitions in the computing literature (Jogalekar and Woodside, 1997; Stephens and Poess, 2004; Law, 1998), but are too subjective to be of use. Somewhat better are the following:

> *"An architecture is scalable . . . if it has a . . . linear (or sub-linear) increase in physical resource usage as capacity increases . . ."* (Brataas and Hughes, 2004)

> *"An algorithm is scalable if the level of parallelism increases at least linearly with the problem size. An architecture is scalable if it continues to yield the same performance per processor, albeit used on a larger problem size, as the number of processors increases."* (Quinn, 1994)

> *"A desirable form of scalability is a resource cost that is at most linear in some measure of performance or usage."* (Messerschmitt, 1995)

Such heuristics might seem reasonable at first blush, but it is normally easy to think of exceptions. Looking for an increase at most linear in resource usage as demands on the system increase is a commonly found rule of thumb (Bondi, 2000; Sun and Rover, 1994). However, linearity is not always necessary or even adequate. Arllit et al, for example, describe the scalability of a large Web-based shopping system and argue that the uncertainty of the workload and fluctuation in capacity requirements makes linearity unattractive (Arlitt et al., 2001). Conversely, consider also the quicksort algorithm, backbone of many scalable information processing systems, which has super-linear time complexity of $O(n \log n)$ in the average case and $O(n^2)$ in the worst case (Rivest and Leiserson, 1990).

Another tempting rule of thumb is to avoid solutions that result in an exponential use of resources in relation to the input size. Such rule would find its roots in the intractable nature of problems that can only be solved by algorithms having exponential time or space complexity (Hopcroft et al., 2006).

Nevertheless, such a statement would represent yet another arbitrary heuristic. In rare cases where the scaling aspect is highly unlikely to cross a given threshold, an exponential growth may be perfectly acceptable.

Our literature review suggests that, so far, attempts to generalize the term *scalability* represent an intuitive ideal. Notably, when an accurate use is found in the literature, it is invariably because of its narrow meaning, as in the case of the parallel computing area (Luke, 1993; Kumar and Gupta, 1994). This limitation has been recognized by some authors, who stated that scalability has dimensions and have tried to characterize them. For example, Gustavson classifies scalability dimensions as performance, economic, physical, addressing, software transparency, communication ability, and technology independence (Gustavson, 1994). Bondi defines three types of scalability: load, space and space-time scalability (Bondi, 2000). Finally, Brataas and Hughes divide scalability into processing, information and connectivity capacity (Brataas and Hughes, 2004). The problem is that scalability is so dependent on the application domain and the system's goals that accommodating all dimensions in pre-defined categories is very challenging, if not impossible.

1.1.3 An Often Overlooked Quality

Our conversations with companies from different sizes and industries suggest that developers often ignore scalability during the system development (Industry Interviews, 2007). This is not surprising. Our extensive literature review (see Chapter 2) shows that few authors are concerned with the precise meaning of the term. In fact, scalability is not even to be found in the some popular software quality taxonomies (Boehm et al., 1976; Keller et al., 1990; McCall et al., 1977; Sommerville, 2004; Kruchten, 1999; Dromey, 1996; ISO 9126, 1991). Many works on scalability refer to narrow classes of systems or technologies, not being applicable to software systems in general. Others are targeted at broader classes of systems, but nearly all are concerned with performance metrics only. Scalability, however, relates to multiple system qualities (see Chapter 3). Furthermore, a number of works judge a system's scalability based on the values assumed by metrics of interest, rather then considering the satisfaction of the stakeholders with these values (i.e., the system scalability goals). Finally, virtually no work addresses the elaboration of scalability requirements.

Scalability is a multi-dimensional problem that cannot be explained by individual technologies or user trends. Its multi-dimensional nature makes it difficult for developers to reason about scalability and build scalable systems that accommodate all the relevant forces at work. Our conversations with developers indicate that they regularly failed to identify these forces and predict all the system trends (Industry Interviews, 2007).[1] Even when they claimed to have considered scalability, they had generally catered only for the extreme cases of environmental and design characteristics. Doing so is a popular fallacy, as extreme cases will often change. The same product may also be deployed at different client sites, each one with its own local limits. More importantly, extreme cases are no basis for understanding the system behavior on other points of the execution range. Take, as an example, a commercial system that uses parallel computation to perform batch processing of data. If the system behavior is known only

[1]Common causes of scalability problems are discusses in Section 4.2.

for the current maximum batch size, little can be said about its scalability. If the size of the data batch is dramatically reduced, the overhead of the parallel algorithm may compromise the users' perceived performance. This system may also not perform at its best for the sub-range of the market that generates most of its company's revenue. Finally, no estimation can be made regarding its behavior in the case of a future increase in the data batch size.

This lack of a systematic treatment of scalability results in problems that often became apparent only when the system is exposed to full load during the production phase, that is, where fixes are costliest according to Boehm's landmark paper (Boehm, 1976)[2]. This scenario may lead to hard-to-maintain workarounds and, on occasions, complete re-designs (Industry Interviews, 2007).

1.1.4 An Increasingly Important Quality

Historically, developers have been relying on hitherto continuous improvements in hardware technology to rescue them from poor software design decisions. In fact, some of the developers we interviewed firmly believed that current technologies, such as multicore CPUs or virtualization, would ensure the scalability of their systems (Industry Interviews, 2007). Nevertheless, according to an internationally respected group of scientists looking into the future of science towards 2020, computing technology is running up against fundamental physical limits:

> *"We postulate that most aspects of computing will see exponential growth in bandwidth but sub-linear or no improvements at all in latency. Moore's Law will continue to deliver exponential increases in memory size but the speed with which data can be transferred between memory and CPUs will remain more or less constant and marginal improvements can only be made through advances in caching technology. Likewise, Moore's law will allow the creation of parallel computing capabilities on single chips by packing multiple CPU cores onto it, but the clock speed that determines the speed of computation is constrained to remain below 5 GHz by a thermal wall. Networking bandwidth will continue to grow exponentially but we are approaching the speed of light as a floor for latency of network packet delivery. We will continue to see exponential growth in disk capacity but the speed with which disks rotate and heads move, factors which determine latency of data transfer, will grow sub-linearly at best, or more likely remain constant. Thus commodity machines will not get much faster . . ."* — The 2020 Science Group (2006)

Consequently, developers will have to take a proactive approach to scalability, designing systems that will take advantage of the available technologies to meet their scalability goals, without running up against economical or fundamental physical limits.

[2]Since the paper was published, improved software processes and technologies have reduced the amount of rework needed to fix problems (Boehm, 2006). Lately, agile methods promised to flatten the curve presented in Boehm's paper (Beck and Andres, 2004). However, data indicates that this flattening does not take place for larger projects (Elssamadisy and Schalliol, 2002; Boehm, 2006). Therefore, the cost of change at later stages of the software development lifecycle continues to be a problem.

1.2 Running Example

A real-world system is used as an illustrative example and case study subject throughout this thesis: the Intelligent Enterprise Framework (IEF). The system contains over 1,500 Java classes and a third of a million lines of code. The Intelligent Enterprise Framework (IEF) is a financial services platform designed by Searchspace to process large volumes of data, build adaptive profiles of business entities that are represented within the data, and generate automated alerts to notify users of behavior that appears fraudulent. [3] Searchspace's customers are primarily retail, investment banks and other financial institutions, but for the reminder of this thesis we will use the word *banks* to consider this group.

Over the last 10 years, the banking and finance sector has seen a considerable consolidation. The rate of electronic transactions has increased dramatically, analytical methods have become more complex, and system performance expectations have increased. In order to maintain its performance as data volumes have grown, the system design went through three major re-designs in a lifespan of seven years. These extensions involved significant time and effort on the part of developers, which also required a better understandings of its scalability barriers. This is a story familiar to many software systems and development organizations.

1.3 Research Problem and Hypothesis

There are undoubtedly many contradicting notions of scalability. In the literature, we have not found a definition that is equally applicable to a variety of domains. So far, attempts to generalize the term represent an intuitive ideal, and it is easy to imagine situations where their rationale is not valid. As a reflection, the use of the term *scalable* in the literature often implies a desired goal or a completed achievement, whose precise nature is left to the reader's imagination.

While computer scientists share an informal, intuitive (and sometimes contradictory) understanding of scalability, our extensive literature review shows that there are as yet no general, precise techniques for characterizing and analyzing the scalability of software systems. As a result, it difficult to identify and avoid scalability problems, clearly and objectively describe the scalability of software systems, evaluate claims of scalability, and compare claims from different sources.

Our research problem is stated as follows:

The Research Problem *The computing community is lacking (1) a precise definition of the term scalability and (2) a systematic, uniform and consistent treatment of scalability that can be applied across application domains and system designs.*

By treatment, we mean the application of techniques that characterize and analyze the scalability of software systems, allowing developers to identify and avoid scalability problems that would otherwise be overlooked.

[3]This thesis has studies consecutive versions of the IEF, starting from the year 2000. Since then, Searchspace has been bought, being currently known as Fortent, and the IEF has been renamed.For consistency, however, we have decided to continue referring to the company as Searchspace and to the system as the IEF.

To address this problem we have defined the following research hypothesis:

The Research Hypothesis *It is possible to (1) provide a precise definition of scalability that is applicable to any application domain and (2) develop systematic techniques, also applicable to any application domain, to characterize and analyze the scalability of software systems, so that developers can identify and avoid scalability problems that would otherwise be overlooked without the definition and techniques.*

By systematic we mean techniques that follow precisely defined steps, and provide the necessary support information, to guide the analyst in the characterization and analysis of software systems scalability. Proving that such techniques can be applied uniformly and consistently across different application domains and system designs is difficult in the lifetime of a PhD. Our aim is therefore to create a definition of scalability and a framework that uses no application domain specific terms and makes no assumptions on the system design. This hypothesis is tested in a small but representative sample of real-systems, as described in Chapter 7.

1.4 Solution and Thesis Contributions

Furthermore, in order to resolve the lack of techniques for characterizing and analyzing scalability, we provide a framework *for the systematic treatment of scalability, including techniques for (a) specifying scalability goals, (b) generating and evaluating strategies to satisfy these goals, and (c) characterizing, analyzing and comparing the scalability of software systems.* The framework characterizes scalability in terms of independent and dependent variables, and objective functions. These variables and functions are derived from a goal model of the system, developed using the techniques in points (a) and (b) above.

The contributions of this thesis are as follows:

- *A uniform and precise definition of scalability that is independent of the application domain and system design.*

  For reasons we will explain in Chapter 3, we define scalability as "the ability of a system to satisfy quality goals to levels that are acceptable to its stakeholders when characteristics of its application domain and design vary over expected ranges". Unlike other definitions in the computing literature that relate scalability with particular metrics (e.g., throughput), scaling characteristics (e.g., number of users), or scaling behavior (e.g.,linear), our definition describes scalability in terms of quality goals and scaling characteristics of the application domain and the system design. These are concepts common to all software systems, resulting in a uniform and precise definition that transcends application domains, system designs and specific system concerns.

- *A framework and a method for characterizing and analyzing software systems scalability that is independent of the application domain and system design.*

  A framework, based on Experimental Program Analysis (EPA) that reveals the impact of scaling characteristics of the system design on the satisfaction of the system quality goals. The framework combines a technique for elaborating scalability requirements with a technique for characterizing and analyzing scalability. A method defines a systematic way to derive from a goal model, the variables and functions to be used in the scalability analysis. As with the definition, the framework and method are based on concepts that are independent of application domain and system design.

- *A technique for describing, modeling and analyzing scalability requirements, and the description at the goal level of common strategies to resolve scalability obstacles identified during requirements engineering.*

  The technique extends the KAOS[4] framework, allowing one to identify, assess and resolve scalability risks systematically during requirements engineering. The result is a consolidated set of requirements in which important scalability issues have been anticipated through the precise, quantified specification of scaling assumptions and scalability goals, two novel concepts introduced in this thesis. The technique has the following values: generates a complete set of scalability obstacles, identifies non-obvious scaling characteristics in the application domain, accurately models the impact of scalability obstacles , and creates a range of alternative resolutions for scalability obstacles, among others.

- *An analysis technique for evaluating and comparing scalability characteristics of software systems that is independent of the application domain and system designs.*

  An analysis technique uses preference and utility functions to quantify the satisfaction of the stakeholder with specific quality goals and the overall system, respectively. By plotting the utility curve against the scaling characteristics of the system, it is possible to evaluate and compare quantitatively the scalability of software systems.

- *Significant case studies demonstrating the applicability of our scalability definition, framework, requirements engineering techniques and analysis technique.*

  Three case studies demonstrated the applicability of the concepts and techniques described in this thesis in a large, complex, proprietary system. The studies were performed from within a company, involving real stakeholders, and facing a number of the difficulties normally found in industrial projects. The report of such experience and the possible research directions resulted from it are themselves relevant contributions to software engineering.

Additional benefits of this research include the following:

- a precise and uniform vocabulary that stakeholders can use to articulate scalability concerns;
- raise awareness of scalability as a quality that can be analyzed with respect to other system qualities;
- counter-arguments to common misconceptions on scalability;

[4]KAOS stands for "Keep All Objectives Satisfied".

- a framework that encourages a proactive approach to scalability when building systems, particularly with respect to scalability problems caused by exceptional application domain scenarios;
- a requirements elaboration technique that can be used as input to various methods for analyzing aspects of software scalability at architectural level, and that can have different uses during the development lifecycle;
- recasted examples from the literature in terms of the scalability framework;
- a number of industry interviews with professional developers describing real scalability problems, with reflections on the development mistakes and lessons learned; and
- Scientific publications in first-class, peer-reviewed conferences (Duboc et al., 2006, 2007, 2008). A journal paper, covering the scalability goal-obstacle analysis technique and the full IEF case study, is currently being written. An industry-oriented article is also planed.

1.5 Scope of this Work

This work concentrates on a definition for scalability, a technique for elaborating and satisfying scalability goals, and an analysis method for quantifying and comparing scalability of software systems. Scalability, however, is a large area of research. For this reason, a number of interesting topics had to be left out of scope:

- We have not investigated techniques for extrapolating the analysis results. As will be explained in Chapter 4, our analysis combines preferences over dependent variables in a utility value. Extrapolating from this utility is challenging because of the different nature of the dependent variables. Leaving the extrapolation of the analysis results out of the scope of this research imposes certain threats to our scalability framework, as it will be discussed in Chapter 4. Extrapolation is considered as future work in Chapter 8.
- We have not explored the design of test cases for scalability. Goal models are a good source to design a representative set of scalability tests. This subject is also discussed in Chapter 8.
- Another area we did not cover is the treatment of uncertainty in both the articulation of requirements and in the analysis technique. Incorporating uncertainty into the scalability framework is one of our priorities for future research.
- Theoretically the scalability framework can be used as input for existing modeling/analysis techniques, such as queuing networks, ATAM, QSEM. Nevertheless, it was out of the scope of this work to create such models/analysis from our framework instantiation.
- We have decided not to develop a catalog of system qualities and respective metrics commonly affected by scalability. In the lifetime of a PhD, a comprehensive catalog would be very challenging to investigate, describe and evaluate across the whole range of architectural styles or software domains. Nevertheless, others could use our framework and method as a starting point to develop

such a catalog. Different instantiations of the framework can be derived from the goal model of a system fitting architecture style and application domain and generalized for that style or application domain.

- We describe at the goal level some of the commonly-used strategies for the design of scalable systems, so that they can be considered during the requirements engineering phase. It is outside the scope of this research to provide a detailed discussion of the advantages and disadvantages of such strategies. The computing literature has a wide range of formal and informal discussions on the subject (Jacobs, 2005; TCSC, 2008; Barra et al., 2001; Yuan and Sharp, 2004; Menascé, 2003; Bondi, 2000; Scholz and Rouvoy, 2007; Ghemawat et al., 2003).
- It is not our intent to provide an out-of-the-box solution, or automated tools for scalability analysis, or predefined formulas, or rules-of-thumb to judge the scalability of a system.
- Finally, it is out of the scope of this research to investigate the scalability of software development techniques and processes.

A number of these points are discussed as future work on Chapter 8.

1.6 Clarifications and Assumptions

Clarifications

Ian Sommerville defines a *stakeholder* as any person or group that will be affected by the system, directly or indirectly (Sommerville, 2004). Stakeholders typically include strategic decision makers, managers of operational units, domain experts, operators, end-users, developers, sub-contractors, customers, certification authorities and so forth (van Lamsweerde, 2008). In this thesis, we refer to "the stakeholder" when it is not relevant to the discussion what is the exact role that stakeholder is playing. However, for ease of argumentation in some points of the thesis, we refer to two specific roles: developer and analyst. *Developers* are people actively involved in the construction of the system, such as business analysts, software architects, programmers and testing analysts. The *analyst* is the person, or group of people, performing the scalability analysis. Multiple roles can be played by the same person.

Chapter 5 describes a technique for elaborating scalability requirements. We chose to use the term *requirements elaboration* because the technique helps to identify which scalability-related information needs to be elicited during requirements engineering, and covers the description, modeling and analysis of the acquired knowledge.

Assumptions

This research relies on a number of assumptions. While we recognize that these assumptions will not always hold in the real wold, we feel that they were reasonable in order to develop the techniques described in this thesis. Furthermore, our case studies demonstrate that these techniques can already provide valuable support for characterizing, analysing and comparing the scalability of software systems. Naturally, we intend to continue the research, so that the framework can be used in a progressively imperfect world. For the time being, the assumptions we rely on are as follows:

- The analyst has access to the stakeholders, documents and data required to construct the goal model.
- The information to characterize scaling ranges and distributions over scaling ranges can be elicited from the stakeholders, documents and tests.
- There is no uncertainty in relation to the ranges and distribution of scaling characteristics.
- There ultimately will be a running version of the system, possibly a prototype, that can be used for scalability analysis.
- There are sufficient data and hardware infrastructure to run the system (or a prototype) over the full range of scaling characteristics for purposes of analysis.
- The metrics being collected for the analysis are a fair representation of the system qualities of interest.
- The analyst has the analytical skills to cater for dependency between analysis variables and for choosing an appropriate utility function.
- The analyst has the ability to choose a representative set of system designs (or system configurations) and data for running any required scalability test.
- KAOS is used as the requirements elaboration technique for the system under scalability analysis. That is, a standard goal model is available for the analysis of scalability at the goal level.

1.7 Research Method

This research combined practice and theory, following cycles of observing, planning, experimenting and reflecting. The research problem emerged from the intuition, observation and experience of the people involved and was confirmed by a comprehensive literature review and discussions with other computer scientists. Knowledge on the subject was built (1) analytically, through the review of the state-of-art in scalability research and related areas, and through courses offered at universities, industry and at conferences; and (2) empirically, through constant experimentation, numerous informal conversations and frequent exposure of ideas to other researchers and practitioners. Research ideas were validated in an industrial setting from the very beginning. Constant experimentation allowed observation, reflection and generation of new ideas.

1.8 Thesis Outline

Chapter 1 has described the motivation and scope of our work. We have also included the research problem and hypothesis, the thesis contributions and assumptions. The remaining chapters are organized as follows:

Chapter 2 covers the background concepts required to understand our work, such as experimental program analysis and goal-oriented requirements engineering. Our position with respect to related work is discussed.

Chapter 3 revisits our definition of scalability and presents a high-level overview of the framework we created for the characterization and analysis of software systems scalability.

Chapter 4 describes the elements of the framework in greater detail and introduces an analysis technique that characterizes and compares the scalability of software systems with respect to the stakeholders' goals.

Chapter 5 presents techniques for elaborating scalability requirements, based on a goal-oriented requirements engineering method. These techniques allows one to anticipate and resolve scalability risks at the goal level.

Chapter 6 brings together our techniques for scalability analysis and for the elaboration of scalability requirements. A method describes the activities required for characterizing and analyzing the scalability of software systems, which includes the systematic derivation of the variables and functions to be used in the scalability analysis from the system's requirements.

Chapter 7 describes a series of case studies using a real-world data analysis system and the application of our techniques to a number of smaller examples taken the literature and from industry interviews.

Chapter 8 summarizes and evaluates the contribution of this work to software engineering and explores future directions of this research.

Appendix A lists a number of definitions of the term *scalability* taken from the scientific and informal computing literature.

Appendix B summarizes a number of interviews carried out with stakeholders and developers in industry about scalability problems they have faced in their companies.

Appendix C contains the design of a comparative case study for the analysis of scalability during the development lifecycle, including its objectives, hypotheses and risks.

General Remarks The *critical evaluation* is discussed on a chapter by chapter basis. Also, many of the remarks in this thesis are the result of our experience as researchers, software developers, project managers and experimenters. We have also undertaken a number of interviews with seasoned stakeholders and developers who have faced scalability problems in their systems. The interviews covered 16 companies of different sizes, industries and maturity level. Each interview was an in-depth discussion of their scalability problems, development processes and lessons learned. While these results were not gathered by any scientific means, it helped us to confirm intuitions, articulate common concerns regarding scalability and propose ideas for the scalability framework. When a remark originating from this tacit knowledge is made in the thesis, we will reference it as "(Industry Interviews, 2007)". A summary of the industry interviews is given in Appendix B.

1.9 Summary

In this chapter, we have discussed the motivation, the research problem and hypothesis, as well as the scope, contributions and assumptions of the work described in this thesis.

Scalability is a frequently asserted attribute of software systems that is rarely validated in any meaningful, systematic way. Inconsistent views on the meaning of the term and the lack of systematic, uniform and consistent techniques for characterizing and analyzing the scalability of software systems make it difficult to identify and avoid scalability problems, to describe clearly and objectively the scalability of software systems, to evaluate claims of scalability, and to compare claims from different sources.

The computing community is lacking a precise definition of scalability and a systematic, uniform and consistent treatment of scalability that can be applied across application domains and system designs. This work aims to provide such a definition and develop systematic techniques to characterize, analyze and compare the scalability of software systems.

Scalability is a large research area and, for this reason, a number of interesting topics had to be left out of the scope of this research. This includes the extrapolation of scalability analysis results, the design of test cases for scalability, the treatment of uncertainty in both requirements elaboration and scalability analysis techniques, among others. This work also relies on a number of assumptions that, although will not always hold true in the real-world, were considered reasonable to enable the development of techniques that provide valuable contributions for the field of software systems scalability.

In the next chapter, we review some background concepts that are required to understand this research and discuss related work.

Chapter 2

Literature Review

Scalability is a large area of research. In this chapter, we provide the theoretical grounding required to understand the scalability framework and give an overview the state-of-the-art in scalability research and related fields.

2.1 Background

This section reviews the concepts of experimental program analysis and goal-oriented requirements engineering, two important techniques that form the theoretical grounding for this thesis.

2.1.1 Experimental Program Analysis

The scalability framework is adapted from the *Experimental Program Analysis* (EPA) technique, a formalization of guidelines and methodologies of scientific experimentation in program analysis (Ruthruff et al., 2006). This technique is defined as follows:

Experimental Program Analysis

> *The evolving process of manipulating a program, or factors related to its execution, under controlled conditions in order to characterize or explain the effect of one or more independent variables on an aspect of the program.* (Ruthruff et al., 2006)

In scientific experimentation, *manipulations* are purposeful changes to *independent variables* in an experiment. In the context of EPA, they are modifications in concrete representations of a program (e.g. source code) or factors relating to its execution (e.g. input or program states). Independent variables manipulations are called *treatments*, whose effects are inferred from *observations* in the experiment.

EPA techniques follow six distinct (and often interleaving) activities:

1. *Recognition and statement of the problem*: Guides and sets the scope for the experimental program analysis activity. It defines specific questions (the problem) and the aspect of the program the experiment will draw conclusions about (the population).

2. *Selection of independent and dependent variables*: Defines the aspect of the program to be manipulated by the EPA technique, a construct with which to measure these manipulations, and the factors for which the experiments do not account, but that could influence or bias observations.

3. *Choice of experiment design*: Determines specific levels from each independent variable's range at which to instantiate treatments; formulates hypotheses about the effects of the treatments on the aspect of the program of interest; and samples a set of elements from the population, assigning treatments to samples.

4. *Execution of the experiment*: Obtains the set of observations on the sample units that measure the effect of independent variables manipulations (treatments).

5. *Analysis and interpretation of data*: Analyzes the collected observations to evaluate hypotheses and determines whether further treatments need to be evaluated or different experimental conditions need to be explored.

6. *Conclusions and recommendations*: Draws conclusions based on experimental program analysis results and, if appropriate, recommends future courses of action.

Chapter 3 describes how the scalability framework addresses these stages.

2.1.2 Goal-Oriented Requirements Engineering with KAOS

Goal orientation is a recognized paradigm for elaborating, structuring and analyzing software requirements (Chung et al., 1999; van Lamsweerde, 2008). In KAOS, which stands for "Keep All Objectives Satisfied", a *goal* is a prescriptive statement of intent the system should satisfy through the cooperation of agents. *Agents* are active system components such as humans, devices, and software. *Domain properties* and *assumptions* are statements about the application domain. The former are descriptive statements about this world. Physical laws are typical examples of domain properties. Domain assumptions are statements to be satisfied by the environment. They are specialized in *expectations* (prescriptive statement to be satisfied by a single environment agent) and *hypotheses* (descriptive statements satisfied by the environment and subject to change).

Goals, domain properties and assumptions can be organized in AND/OR refinement structures. An *AND-refinement* relates a goal to a set of subgoals; this means that satisfying all subgoals in the refinement is a sufficient condition in the domain for satisfying the goal. *OR-refinement* links relate a goal to a set of *alternative* AND-refinements; this means that each AND-refinement represents an alternative way to satisfy the parent goal. Terminal goals in a goal-refinement structure are goals that are assigned to some individual agent who is alone responsible for guaranteeing the satisfaction of the goal (Feather, 1987). A *requirement* is a terminal goal under the responsibility of the software-to-be; an *expectation* is a terminal goal under the responsibility of an agent in the environment (van Lamsweerde, 2008).

A goal specification includes a *name*, a *category*, a natural language *definition*, and an optional *formal definition* expressed in the Metric Linear Temporal Logic (Koymans, 1992). We use the following standard temporal logic operators in this thesis: $\Box P$ (P is true in all future states), $\Diamond P$ (P is true in some future state), and $\Diamond_{\leq d} P$ (P holds in some future state that is at most d time units from the current state). The natural language and formal definition of a goal define what it means for the goal to be satisfied in an absolute sense. Partial levels of goal satisfaction can be specified in precise terms using domain-specific *quality variables* and *objective functions* (Letier and van Lamsweerde, 2004). Objective functions are used for evaluating and selecting among alternative system designs. As an example, take the specification fragment below published by Letier and van Lamsweerde (2004). It describes a goal for the intervention of an ambulance in a urgent incident for an ambulance dispatching system:

Goal 2.1

Goal Achieve [Ambulance Intervention]

Category Performance

Def For every urgent call reporting an incident, an ambulance must arrive at the incident scene within 14 minutes.

Formal Def $\forall$ inc:Incident

Reported (inc) $\Rightarrow \Diamond_{\leq 14'}$ ($\exists$ Ambulance:Ambulance) Intervention(amb, inc)

Quality Variable: RespTime

Def: time between first report of the incident and arrival of the first ambulance at the incident scene.

Sample Space: Set of reported incidents.

Objective Functions:

Name	Definition	Mode	Must
8MinRespRate	Pr(RespTime $\leq$ 8')	Max	50%
14MinRespRate	Pr(RespTime $\leq$ 14')	Max	95%

The natural language and formal definitions state that this goal is satisfied in an absolute sense if an ambulance arrives at the incident scene within 14 minutes. The quality variable RespTime is a *random variable* whose sample space is the set of reported incidents and whose value denotes the time for an ambulance to arrive at the incident scene. The objective functions state that the system should be designed to maximize the *probability* of an ambulance arriving in less than 8 or 14 minutes. The "Must" column indicates that in at least 50% of the cases an ambulance should arrive at the incident scene within 8 minutes and in at least 95% of the cases it should arrive within 14 minutes. This high-level goal is not assigned to any agent and should be refined further.

Goal-obstacle analysis consists of taking a pessimistic view of the model elaborated so far. An *obstacle* to some goal is an exceptional condition that prevents the goal from being satisfied (Potts, 1995; van Lamsweerde and Letier, 2000). An obstacle O is said to *obstruct* a goal G in some domain characterized by a set of domain properties Dom iff (1) the obstacle entails the negation of the goal in the domain (i.e., O, Dom $\models \neg$ G), and (2) the obstacle is not inconsistent in the domain (i.e., Dom $\not\models \neg$ O). The principle of obstacle analysis consists of systematically identifying as many obstacles as possible to the satisfaction of goals, and finding ways to resolve the identified obstacles by producing a more complete set of goals and requirements to prevent, reduce, or mitigate the identified obstacles. The selection of a preferred resolution depends on the obstacle likelihood and critically, and the impact of the resolutions on the satisfaction of high-level goals (Letier and van Lamsweerde, 2004).

Graphically, goals are organized in tree structures, as in Figure 2.1. Goals and assumptions are represented by parallelograms. The trapezoid is used for representing domain properties or hypotheses. Hexagons are used for agents, and the circle in the line joining a goal to an agent is a responsibility assignment. An AND-refinement is represented by an arrow with a small circle connecting the sub-goals contributing to the refined goal; the latter is the target of the directed link. An OR-refinement is graphically represented by multiple AND-refinement arrows pointing to the same goal. A negated arrow represents a goal obstruction, and a "reverse" parallelogram indicates an obstacle.

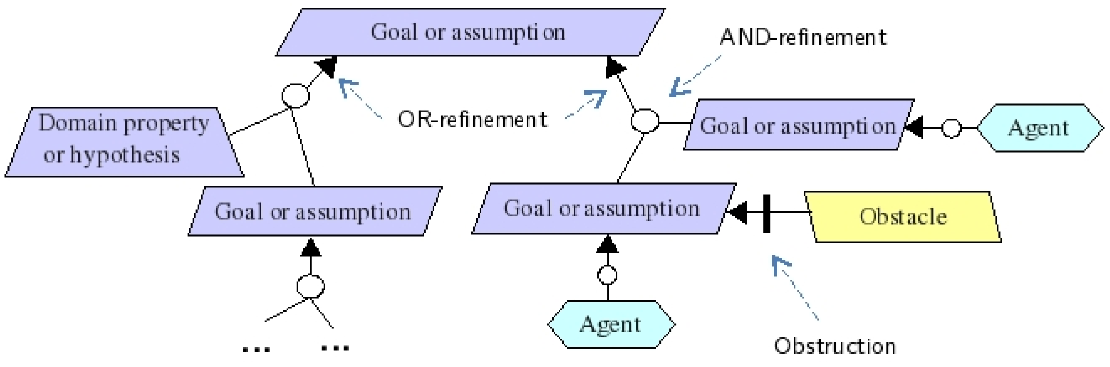


Figure 2.1: KAOS goal model notation.

Goal-oriented requirements elaboration methods such as KAOS and the NFR framework (Chung et al., 1999) provide a rich set of techniques for the incremental elicitation, specification, analysis, and evolution of requirements models. However, as reported in (Duboc et al., 2008), during our first application of KAOS to an industrial system, we could not find any support for the elaboration of scalability requirements. Specifying a high-level goal named Scalable System and gradually refining this goal into more precise requirements is not adequate because of the lack of a precise definition for the description of the high-level goal and the lack of guidance on how to refine it into testable scalability requirements. We tackle this problem in Chapter 5.

2.2 Related Work

This section gives an overview of the state-of-the-art in scalability and other related areas.

2.2.1 The Various Contexts of Scalability

Scalability has been studied in a variety of contexts, such as parallel computing (Luke, 1993; Kumar and Gupta, 1994; Sun et al., 2005), video imaging (North, 2006), hypermedia (Anderson, 1999a; Fielding and Taylor, 2002; Anderson, 1999b), mobile and ubiquitous systems (Deters, 2001; Rana and Stout, 2000; Bivens et al., 2004; Kang et al., 2008), networks and routing protocols (Li et al., 2008; Kosch et al., 2006; Alazzawi et al., 2008), distributed systems (Zhang et al., 2007; Bahsoon and Emmerich, 2008; Kalinov, 2004), gaming (Jiang et al., 2005; Callow et al., 2007; Müller and Gorlatch, 2006), simulation (Ji et al., 2006; Law, 1998; Egea-Lopez et al., 2006), information retrieval (Imafouo, 2005; Monch, 2002), data mining (Lutu, 2002; Buehrer and Chellapilla, 2008; Blockeel and Sebag, 2003), operating systems (Silva et al., 2006; von Behren et al., 2003; Jones et al., 2003), quantum computing (Meter and Oskin, 2006) and software process (Laitinen, 2000; Tepfenhart et al., 2002), among others. We next exemplify how scalability is treated in some of these contexts:

Parallel Systems

One of the areas in which scalability has been extensively studied is parallel systems. The parallel computing community has a few well-established definitions, including *fixed-size speedup* (where the number of processors is scaled in the presence of a fixed workload to reduce processing time), *fixed-time* *speedup* (where processor attributes are scaled so that workload can be scaled to retain a fixed processing time), and *efficiency* (the ratio of speedup to the number of processors) (Luke, 1993; Kumar and Gupta, 1994). Based on these definitions, a number of scalability metrics and functions have been defined, such as *P-scalability* (Jogalekar and Woodside, 1997), *isoefficiency* (Rao and Kumar, 1987; Grama et al., 1993), *scaled speedup* (Gustafson, 1995), *sizeup* (Sun and Gustafson, 1993), and *isospeed* (Sun and Rover, 1994; Sun, 2002). These metrics, related mainly to performance, exploit the uniform nature of parallel systems. In particular, they rely on the fact that such systems are typically symmetric compositions of a single, basic processing node, inducing fairly straightforward closed-form characterizations of their scalability properties. Nevertheless, because of fundamental differences between parallel computing and other classes of systems, these metrics cannot be applied more broadly (Sun et al., 2005; Pastor and Bosque, 2001; Jogalekar and Woodside, 1998). Distributed systems, for instance, typically consist of heterogeneous processors and interconnection mechanisms, irregular topology, concurrent input, geographic separation, etc. Therefore, merely increasing hardware resources may not scale up a system's performance if a bottleneck lies in the software/middleware (Bondi, 2000). Attempts to generalize metrics for parallel systems are discussed later in this chapter.

Model Checking

Another area where scalability has received systematic treatment is model checking. The state space size a model checker can handle can be taken as an indicator of scalability, where being able to handle a larger the state space indicates a more scalable model checker. There is a large body of research that concentrates on tackling the so-called state explosion problem of model checkers (Grumberg and Peled, 1999) through heuristics and optimizations, such as exploiting the symmetry of the state space (Emerson and Sistla, 1997; Clarke et al., 1996; Ip and Dill, 1996). Model checking techniques can then be easily compared through the use of a few common metrics of interest, normally related to resource usage. But such approaches and metrics are inapplicable to systems other than model checker.

Computational Complexity

The term *scalability* is often used to refer to more theoretical notions of computing, as in computational complexity (Goldsmith et al., 2007; Yang et al., 2003). A common intuition is that an algorithm with a polynomial complexity is scalable, while one with an exponential complexity is not scalable (Sastry et al., 2005; Nadeau and Teorey, 2002). In such cases, scalability is being related to the tractability of the algorithm.

Programming Languages and Software Development Processes

Scalability has also been studied in contexts other than software systems. More generally, artifact size (e.g., lines of code, test suite size) is used as a scalability metric to compare software tools and methods. For example, programming language advocates often discuss a language's scalability, where more powerful constructs and features mean a more scalable language (Vanier, 2001; Odersky et al., 2008). In such cases, the term is related to the complexity of the coding tasks, and language scalability is normally achieved through abstraction. Another common concern is the scalability of software development processes, tools and methods. Some software development practices, designed for large departments, are inefficient for smaller ones (Laitinen, 2000), while others do not bring the envisaged benefits in large projects (Elssamadisy and Schalliol, 2002; Boehm, 2006).

2.2.2 Scalability — Definition, Analysis and Prediction

The following studies explicitly tackle the problem of scalability. Some attempt to define the meaning of the term, while others are concerned with analysing and predicting the scalability of software systems. This list of works is not exhaustive. Yet, it highlights the main aspects of the state-of-the-art in scalability research. Some works are presented in more detail in order to convey a more concrete view of the different treatments of scalability. The main distinctions with respect to our framework are discussed.

Scalability Definitions

Like ourselves, other authors have questioned the meaning and use of the term *scalability* and have attempted to provide better definitions (Hill, 1990; Steen et al., 1998; Bondi, 2000; Weinstock and Goodenough, 2006; Sun and Rover, 1994; Luke, 1993; Zirbas et al., 1989; Law, 1998). A list of definitions of scalability is given in Appendix A. However, these earlier attempts to define scalability represent an intuitive ideal or are restricted to a narrow meaning, as discussed in Chapter 1. In particular, as it will be explained in Chapter 3, we associate scalability with the satisfaction of quality goals. A few previous works have related scalability to requirements and/or Quality of Service (QoS), either formally or informally. For example:

> *"Scalability means not just the ability to operate, but to operate efficiently and with adequate quality of service, over the given range of configurations."* (Jogalekar and Woodside, 2000)

> *"Scalability is the ability of a system to continue to meet its response time or throughput objectives as the demands for the software functions increases."* (Smith and Williams, 2001)

> *"The scalability of a system architecture must be seen in the context of the requirements placed on it. It represents the ability to fulfill capacity requirements over some desired range, while continuing to satisfy all other requirements: functional, statistical work-mix, quality of service, unit cost of ownership, etc.. We shall refer to the complete set of requirements other than capacity as the IT profile. An architecture is scalable with respect to an IT profile and a range of desired capacities if it has a viable set of instantiations over that range. Viable will therefore be taken to mean (1) satisfying the requirements of the IT profile, (b) technically realizable and (c) with linear (or sub-linear) increase in physical resource usage as capacity increases over the range. It is factor (c) which is the essence of the scalability concept, with factors (a) and (b) acting as constraints."* (Brataas and Hughes, 2004)

These definitions, however, suffer from the same weakness as the ones discussed in Chapter 1. They are subjective (because adopt terms such as adequate) or are restricted to performance metrics and the idea of linear scalability.

Techniques Adapted from Parallel Computing

With respect to the analysis of scalability, some early papers attempted to adapt scalability notions from parallel computing to other classes of systems. Jogalekar and Woodside define a metric for evaluating scalability based on the power of the system (the ratio between throughput and mean response time) to the cost of obtaining this power, and a method for finding the most economical path for evolving a system (Jogalekar and Woodside, 1997, 2000). More recently, Sun et al. (2005) extended the isospeed scalability metric to an isospeed-e metric that is suitable for both homogeneous and heterogeneous computing. These same authors created a Scalability Testing and Analysis System (STAS) that provides analysis of algorithms and systems based on the isospeed-e metric (Chen and Sun, 2006). Kalinov (2004) adapts metrics proposed for linear algebra libraries to heterogeneous platforms.

These papers, which are mainly targeted at distributed systems, also concern *performance* metrics only. We view scalability as a quality that relates to a number of system qualities, not only performance, as reported in (Duboc et al., 2007). Therefore, in our framework, scalability may be analyzed with respect to metrics related to other system qualities as well. In fact, the metrics used by these papers can be used in the scalability framework to quantify dependent variables.

Techniques Targeted at Narrow Classes of Systems and Specific Technologies

More recent work has targeted narrow classes of systems and technologies (Kwok and Wong, 2008; Coarfa et al., 2007; Cecchet et al., 2002; Kim and Ellis, 2001). Papavassiliou et al. (2002), for example, describe a simulation methodology to evaluate the scalability of ad-hoc mobile networking technologies. Liu et al. (2002) describe an empirical investigation into the scalability of the Enterprise JavaBeans (EJB) technology. Kong et al. (2006) concentrate specifically on peer-to-peer (P2P) systems based on distributed hash tables (DHT) and present a technique for characterizing the scalability of these systems under random failures.

The advantage of using such techniques is that they offer off-the-shelf analyses of scalability problems that are specific of a particular class of systems and technologies. Also, by exploiting characteristics of these systems and technologies they might be able to accomplish a more precisely prediction of system qualities of interest. However, by applying only such techniques, one may overlook problems that are specific to the system being analyzed. Our framework, in contrast, aims to establish a uniform notion of scalability that can be applied in a wide variety of application domains and system designs. Yet, it is based on a systematic requirements engineering technique that will help to identify and resolve scalability risks that are particular to the system being analyzed. Finally, one may create specific instantiations of the scalability framework for classes of systems and/or technologies that will act as a first guidance to characteristics to consider for system fitting these categories. Such instantiations could be extended by using the systematic requirements engineering techniques to include other aspects that are specific to the system under analysis.

Techniques for Evaluating Scalability with Respect to Performance

Some papers in the computing literature are concerned with analyzing the scalability of broader classes of software, but with respect to performance metrics only. For instance, Steen et al. (1998) describe a systematic approach to guide the distribution and application of scaling techniques, such as replication and caching, based on performance metrics. Masticola et al. (2005) focus on the inception phase of the Rational Unified Process (RUP), estimating scalability from user scenarios, experimentation, public study data, and performance data from a baseline system. Some other examples are given in more detail below:

Performance Non-Scalability Likelihood

Weyuker and Avritzer define a Performance Non-Scalability Likelihood (PNL) metric, whose goal is to determine whether a system can continue to function with acceptable performance when the workload has been significantly increased (Weyuker and Avritzer, 2002). The metric is calculated with the formula $PNL(P, Q) = \sum_{s} Pr(s)C(s)$, where $Pr(s)$ is the probability of a given state s and $C(s)$ is the acceptability of a certain performance: $C(s) = 1$ if performance is acceptable and $C(s) = 0$, otherwise. The process to apply the PNL metric is the following: (1) define the performance objective of the system; (2) determine the appropriate distribution of arrival processes and service times; (3) model the system as a queuing network; (4) solve the model to generate the state probability distribution of the system; and (5) compute the PNL metric. The process requires the collection of significant amounts of field data to derive the operational distribution, including an operational profile of the system under study.

In common with our framework, their paper considers the acceptability of the system's performance objectives when evaluating scalability. However, it offers no guidelines on how to define these objectives, which in the paper are described only in terms of the maximum acceptable response time. Furthermore, unlike our framework, this paper is restricted to queuing network models and performance metrics.

Brataas and Hughes's Architectural Scalability

Brataas and Hughes (2004) explore the scalability of an architecture through measurement and a combination of static and dynamic models. The authors consider an architecture to be scalable, over a particular set of requirements, if the physical resource usage per unit of capacity remains roughly constant. A static modeling technique, the Structure and Performance (SP) specification method, defines the services used by each software component in the form of a work-complexity matrix (Jogalekar and Woodside, 1998). This static model determines the resource demands of the system, which are then used in a queuing network to model the dynamic behavior of the system.

The method consists of the following steps: (1) determine the scaling objectives (i.e., the design space to be explored); (2) construct static and dynamic models; (3) validate the baseline model by measurement,; (4) explore scalability by increasing capacity; and (5) validate scalability projections (where possible).

This work is interesting because it considers scalability with respect to the system requirements. According to the authors, a system is scalable if it can fulfill capacity requirements over some desired range, while continuing to satisfy all other requirements: functional, statistical work-mix, quality of service, unit cost of ownership, etc. These other requirements are referred to as *the IT profile*. The paper, however, has a limited view of scalability, only considering scalable a system that can satisfy its IT profile while maintaining a *linear* (or *sub-linear*) *increase* in physical resource usage as capacity increases over the range. We dispute the use of heuristics such as "linear scalability", which may impose an unnecessary demand on the system. We argue, instead, that scalability objectives should be derived from high-level business goals. Furthermore, Brataas and Hughes' work is also restricted to queuing networks and performance metrics.

D'Antonio et. al's Scalability of Component-based Frameworks

This paper (D'Antonio et al., 2004) proposes an engineering approach to the study of scalability in distributed systems. The solution consists of developing a model of the orchestrated behavior of a system's components, evaluating the system performance, and identifying bottlenecks. More precisely, the solution starts by defining a set of *independent parameters* for the model (e.g., number of users, average number of requests per user, number of user input components, etc.) and evaluating how the number of messages exchanged in the system varies relative to such parameters. The system available resources are then modeled with a queuing network. Message arrival rates are defined as functions of the independent parameters, and service rates are estimated based on the message processing time. Finally, the authors replace independent parameters with realistic scenarios, varying one parameter at a time, and estimating the order of magnitude and trends of *dependent variables* (e.g., message arrival time and server's service rate).

Scalability analysis on our framework could theoretically be conducted in similar fashion: by using a queuing network to estimate trends of dependent variables based on the variation of independent variables. However, our analysis goes beyond the estimation of values of dependent variables, taking into consideration the satisfaction of the stakeholder with the values assumed by these dependent variables (described by preference and utility functions). Furthermore, our analysis is not limited to performance metrics or any particular type of architecture. It establishes, instead, a uniform notion of scalability that can be applied in a wide variety of application domains and system designs.

Williams and Smith's Quantitative Scalability Evaluation Method (QSEM)

Williams and Smith describe QSEM, a model-based approach to evaluating quantitatively the scalability of Web-based applications and other distributed systems (Williams and Smith, 2005). QSEM uses straightforward measurements of maximum throughput for different numbers of processors or nodes, and extrapolates the results to a higher number of processors or nodes. The method consists of seven steps: (1) identify critical use cases; (2) select representative scalability scenarios; (3) determine scalability requirements; (4) plan measurement studies; (5) perform measurements; (6) evaluate data; and (7) present results.

Use cases and scenarios chosen in steps 1 and 2 are the ones that represent the typical uses of the application or that have high resource demands, and therefore contribute to the dominant load. The scalability requirements, defined in step 3, should be precise, quantitative and measurable. They are often described by combining performance requirements, performance workload, and projected growth. In order to plan the measurement studies, in step 4, it is necessary to have quantitative information about typical execution paths, thinking time between user requests, typical database queries and the probability of taking different execution paths. In step 5, measurement experiments are conducted and regression analysis is used to determine which of the scalability models best describes the application's observed behavior, which are then used to extrapolate the results. Finally, steps 6 and 7 evaluate the measurement data to determine whether the scalability requirements can be met and then select the best scaling strategy, presenting the results.

This interesting work focuses on scalability with respect to performance, uses measurement, and builds on previously proposed models (linear scalability, Amdahl's law, super-serial model and Gustafson's law) [1]. Its narrower scope makes it well suited to the exploration of throughput with respect to the chosen scalability scenarios. Nevertheless, despite the emphasis on the importance of determining scalability requirements, the authors offer weak guidelines on how to elaborate these requirements, which could lead to overlooking important scenarios. They also do not consider that the level of satisfaction of requirements may vary as quantities in the application domain scale.

Our framework, in contrast, recognizes that scalability problems often come from unexpected system usage and exceptional circumstances, and provides a systematic way to explore these scenarios during requirements engineering. It also formalizes the concepts of scaling assumptions and scalability goals, which describe the "required performance to be achieved expressed in terms of the workload mix and the expected intensity"—an interesting need highlighted by the authors, but not specifically addressed. Finally, our framework also incorporates stakeholders' goals—as preference and utility functions—in the scalability analysis.

Weinstock and Goodenough's Scalability Model

Weinstock and Goodenough (2006) provide an interesting discussion on the scalability of performance-critical systems. The discussion includes their definition of the term, causes of scalability failures, the trade-off between system qualities, and an audit for helping the assessment of proposed scalability strategies. The authors also present initial ideas on a model for scalability.

Weinstock and Goodenough's model looks at adding hardware resources incrementally to satisfy a response metric `M`. The analysis is defined in terms of processor units `P`, and assumes that the system

[1] A discussion of these models can be found in Williams and Smith (2004).

consists of one or more processors added incrementally. The maximum capacity of a single processor is $\mathtt{D}_1$ demands units. That is, a single processor running alone will satisfy the metric $\mathtt{M}$ as long as the instantaneous demand $\mathtt{d}$ is within the processor's capacity ($\mathtt{d} \leq \mathtt{D}_1$). The authors note that there are potential overheads in adding processors and that it may not linearly increase the system capacity or decrease performance at low demand. They thus define *Efficiency* $\mathtt{E}_n$ as a measure of the actual capacity available in the $\mathtt{n}$-processor system when overhead is taken into account, divided by the capacity that would be available if the overhead were ignored: $\mathtt{E}_n = \Sigma_{i=1}^{n} D_i \,/\, nD_1$. Efficiency is used to measure the relative scalability of a system. A system architecture whose efficiency falls off slowly as more capacity is added is more likely to continue to be scalable than one in which efficiency falls quickly.

The model also defines C_i as the cost of extending the system's capacity by adding the i^{th} unit, and a *cost-effectiveness* metric K_i as some function $F(C_i, E_i)$, such that when two scaling strategies have different values of K, it is always more effective to pick the one with higher value when adding the i^{th} unit.

Like our framework, Weinstock and Goodenough's model allows comparing the scalability of alternative system architectures. The model is, however, limited to system performance with respect to the number of processors and does not consider the stakeholder's goals. The authors also tackle the issue of costs (an issue that is not directly addressed by our scalability framework), but they point out that much more work is needed in evaluating the cost-effectiveness of a scalability strategy.

Munsterman's Software Scalability Evaluation Method

Munsterman describes a method for improving software scalability (Munsterman, 2008). The author's main concerns are (1) the system's ability to accommodate more users without modifying its hardware infrastructure; and (2) the portability of the code to distributed servers or multiple projects. The model consists of finding, analyzing and choosing solution strategies for removing bottlenecks at the system's code level.

Various methods are suggested for finding bottlenecks, including load and stress tests, execution information obtained from log files of application servers and bug tracking systems, and experience of the people involved with the application. If multiple bottlenecks are found, they are ranked by priority. The cause and effects of bottlenecks are then analyzed. For such, the author proposes the use of a profiler to measure the behavior of the application at runtime, and the calculation of metrics that measure code reuse (coupling, cohesion, complexity, instability, and documentation rate). Knowledge about the bottleneck is used to solve the scalability problem. The author suggests that this knowledge can be gained through white papers, community forums and best practices.

Munsterman's solution is targeted at the specific problem of improving a system's scalability (with respect to performance) through modifications in the system's configuration and code. The author claims that only after the software is optimized, scalability through hardware should be considered. Although the system designer should indeed aim for the best use of the resources available and try to avoid software bottlenecks, code optimizations are usually one-off scalability strategies only. Munterman's model has, therefore, a different scope from the scalability framework, which aims to characterize, analyze, and compare the scalability of software systems, regardless of the scaling strategy used to achieve the desired scalability. The author also overlooks the scalability requirements of the system and how to evaluate whether the code optimizations will meet these requirements.

2.2.3 Software Architecture Evaluation

The vast majority of works we reviewed in the area of scalability analysis and prediction are concerned with performance metrics. We, however, view scalability as a quality that relates to other system qualities, as will be explained in Chapter 3. For this reason, our work also relates to the area of architecture evaluation methods.

This body of research is concerned with the analysis of software qualities at the architectural level. Some well-known methods in this area are: the Scenario-based Architecture Analysis Method (SAAM) (Kazman et al., 1996), the Architecture Tradeoff Analysis Method (ATAM) (Kazman et al., 1998), the Cost Benefit Analysis Method (CBAM) (Kazman et al., 2001), the Scenario-Based Architecture Reengineering (SBAR) (Bengtsson and Bosch, 1998) and the Performance Assessment of Software Architectures (PASA) (Smith and Williams, 2001). These methods generally identify the quality goals of interest and then evaluate the strengths and weaknesses of the architecture to meet the desired goals. Depending on the method, the evaluation explicitly addresses a single quality or multiple quality goals at the same time. In general, these methods suffer from a considerable drawback: the wrong choice of variables and functions could compromise the analysis results; yet they do not provided guidance on this task. Our scalability framework, in contrast, includes systematic techniques to identify variables and functions from scalability goals.

Before looking into these methods in more detail, we first review some concepts used by them:

Architectural Style: a description of components types and their rules of configuration. It also includes a description of the pattern of data and control interaction among the components and an informal description of the benefits and drawbacks of using that style (Shaw and Garlan, 1996; Buschmann et al., 1996).

Attribute-based Architectural Style (ABAS): provides a foundation for more precise reasoning about architectural design by explicitly associating a reasoning framework (whether qualitative or quantitative) with an architectural style (Klein et al., 1999). It is a pre-packaged set of analyses and questions for the architect, based upon solutions to commonly recurring problems and known difficulties in employing those solutions.

Scenarios: brief narratives of the expected or anticipated use of a system from both developer and end-users viewpoints. They are considered important tools for exercising an architecture in order to gain information about the system's fitness with respect to a set of ordered quality attributes (Kazman et al., 1996).

We now give a short description of some of the well-known architecture evaluation methods:

<u>SAAM</u>

SAAM is a structured method employing scenarios to analyze architectures (Kazman et al., 1996). It elicits stakeholders' input to identify quality goals that the architecture is intended to satisfy, uses scenarios to operationalize these attributes, and indicates places where the architecture fails to meet these requirements. The steps of SAAM are: (1) describe the candidate architecture; (2) develop scenarios; (3) perform scenario evaluations; (4) reveal scenario interactions; and (5) present overall evaluation.

Scenarios are classified as direct (supported by the current architecture) and indirect (requires a modification to the architecture to be satisfied). Scenario evaluations, indicate for each indirect scenario a list of changes that are necessary and the cost of performing these changes. Determining scenario interaction is the process of identifying scenarios that affect a common set of components.

SAAM is concerned with a system architecture's fitness with respect to a number of qualities, including correctness, security, reliability, availability, maintainability, predictability, and scalability[2], among others. Although the scenarios defined in step 2 force developers to consider future uses and changes to the system, there is no specific guidance on how to elaborate the scalability scenarios, nor are these scenarios treated differently from other types of scenarios. In contrast, our scalability framework explicitly addresses the elaboration of scalability requirements and is concerned with characterizing the effect that scaling characteristics will have on multiple system qualities, whether or not a change is required to the system architecture.

<u>SBAR</u>

SBAR is a method for reengineering software architectures (Bengtsson and Bosch, 1998). It receives as input an updated requirements specification and the existing software architecture and then produces an improved architectural design. The process starts with incorporating new functional requirements into the architecture. Quality attributes (QA) are estimated using qualitative or quantitative assessment techniques, and the QA's estimates are then compared to the non-functional requirements. If the requirements are not met by the estimations, the architecture is improved by selecting an appropriate QA-transformation, resulting in a new version of the architectural design that is fed back to the start of the process.

Despite being aimed at reengineering software architectures, SBAR relates to our scalability framework because it assesses multiple quality attributes against non-functional requirements. Four different approaches are used for the evaluation of the architecture with respect to quality attributes: scenarios, simulation, mathematical modeling and objective reasoning. Scenarios are the primary method of assessment. However, no guidelines are given to specify these scenarios, nor is there any mention of scenarios that consider the scaling of domain characteristics. Furthermore, the method assumes that

[2]The authors do not define what they mean by scalability.

requirements specifications exist, as well as the targets for non-functional requirements. The lack of guidelines for both the specification of scenarios and non-functional requirements introduces the risk of overlooking or wrongly estimating the scaling of domain characteristics, which could jeopardize the estimation of quality values.

ATAM

ATAM is a scenario-based technique to assess the consequences of architectural decisions in the light of quality attribute requirements (Kazman et al., 1998). Not only does it review how well an architecture satisfies particular quality goals, but it also provides insight into how these goals interact with each other (Clements et al., 2002). There are three types of scenarios in ATAM:

Use scenarios: describe the typical uses of the completed running system;

Growth scenarios: describe ways in which the architecture is expected to accommodate growth and change in the moderate near term (expected modifications, changes in performance or availability, porting to other platforms, and so on); and

Exploratory scenarios: describe extreme forms of change (such as order of magnitude changes in performance or availability requirements or major changes to the system's infrastructure or mission).

ATAM's steps are as follows: (1) present ATAM to stakeholders; (2) present business drivers; (3) present architecture; (4) identify architectural approaches; (5) generate quality attribute utility tree; (6) analyze architectural approaches; (7) brainstorm and prioritize scenarios; (8) analyze new architectural approaches; and (9) present results.

Step 1 simply describes the ATAM method for all the people involved in the assessment of the system. The business drivers and the system architecture are described in steps 2 and 3 according to templates. For example, the business drives should include a description of the business environment, history, market differentiators, driving requirements, stakeholders, business constraints, technical constraints, and quality attributes. In step 4, architectural approaches and architectural styles are identified as a means of addressing the highest priority quality attributes. The generation of the quality attribute tree in step 5 results in a prioritized list of scenarios. For such, quality factors that contribute to the system "utility" (performance, availability, security, modifiability, etc) are elicited, specified down to the level of scenarios, annotated with stimuli and responses, and prioritized. Prioritization is based on the importance of each node to the success of the system and the degree of perceived risk posed by the achievement of this node. The architectural approaches that address the highly-ranked scenarios are elicited and analyzed in step 6. The output of this phase is a list of architectural approaches, questions associated with them, and the architect's response to these questions (frequently including a list of risks, sensitivity points and trade-offs). This step may also include rudimentary analyses of the quality attributes. This analysis is not meant to be comprehensive and detailed, but commensurate with the level of detail of the architectural specification. A brainstorm with a larger stakeholder community takes place in step 7 for the generation of more scenarios. The highly-ranked scenarios are analyzed in step 8.

ATAM may be perceived as the technique that most overlaps with the scalability framework. It concerns multiple qualities at the same time, it aims for the precise statement of quality attribute requirements, and it uses utility to evaluate the satisfaction of these requirements. Its main objectives are, however, different. ATAM is concerned with assessing the consequences of architectural decisions, with respect to the different system qualities. This assessment may be qualitative. The scalability framework, on the other hand, is concerned with explicitly and quantitatively showing the effect that scaling characteristics in the application domain and system design have on the satisfaction of quality goals—independent of any change in the system architecture. Furthermore, ATAM gives no systematic guidelines to elaborate quality requirements, or growth and exploratory scenarios. As will be discussed in Chapter 5, scalability problems are sometimes the result of unforeseen scenarios. Without a systematic technique, exceptional scenarios are likely to be overlooked. ATAM's use of utility also differs from the notion of utility in our scalability framework. This difference will be described in Section 2.2.5.

CBAM

CBAM is a method for analyzing the costs, benefits, and schedule implications of architectural decisions. CBAM begins where the ATAM concludes and depends on the artifacts produced by ATAM. The method aids in the specification and documentation of costs, benefits and uncertainty of a "portfolio" of architectural investments and gives stakeholders a framework for applying a rational decision-making process that suits their needs and risk averseness (Kazman et al., 2001).

The steps of CBAM are as follows: (1) collate scenarios, both new ones and the ones elicited during ATAM; (2) refine scenarios; (3) prioritize scenarios; (4) assign utility for each quality attribute response level; (5) develop an architectural style for the scenarios and determine their expected quality attributes response levels; (6) determine the utility of the expected quality attributes response levels by interpolation; (7) calculate the total benefit obtained from an architectural style; (8) choose a new architectural style based on return on investment (ROI), subject to costs and schedule constraints; and (9) confirm the results against intuition.

In CBAM, a scenario contains the worst-case, current, desired and best-case response levels of the quality attribute associated with that scenario. In step 3, scenarios are prioritized by allocating 100 votes to each stakeholder and having them distribute the votes among scenarios by considering the *desired* response value of each scenario. A weight of 1.0 is assigned to the highest-rated scenario, and the other scenarios receive weights that are relative to that one. In step 4, a utility is assigned for each quality attribute response level, forming a utility curve. In step 5, the utility of the *expected* quality attributes value is determined by interpolation, using the utility curve. The total benefit is obtained by subtracting the utility value of the "current" level from the "expected" level for each quality and computing a weighted sum using the votes elicited in step 3.

Unlike CBAM, costs have not been thoroughly investigated in our scalability framework. Theoretically, cost is another goal that has to be satisfied and can be mapped to a dependent variable. However, more research is needed in order to incorporate costs into our framework. One advantage of CBAM is that it considers uncertainties, defining a range of values for each cost, response and utility. This is still the subject of future research in the scalability framework. Particularly, we want to capture uncertainties about future values of scaling assumptions in the goal graph, to then incorporate them into the scalability analysis.

Nevertheless, building upon ATAM, CBAM shares the same disadvantages as ATAM with respect to the lack of systematic techniques for elaborating quality requirements and scenarios. Finally, the authors admit that one of the biggest problem with CBAM is that it is difficult for stakeholders to quantify the expected utility level of an architectural style. They also observe that highly variable judgments can result if their interpretations cannot be calibrated with the current system's business goals. In the scalability framework, the problem of quantifying preferences and utility according to business goals is dealt with by using a systematic goal-oriented requirements elaboration technique. CBAM's use of utility also differs from the notion of utility in our scalability framework. This difference will be described in Section 2.2.5.

PASA

PASA is a scenario-based method for performance assessment of software architectures. It uses the principles and techniques of software performance engineering (SPE) to determine whether an architecture is capable of supporting its performance objectives (Smith and Williams, 2001).

The PASA process consists of the following 10 steps: (1) present an overview of the process to stakeholders; (2) present an overview of the architecture; (3) identify critical use cases; (4) select key performance scenarios; (5) identify performance objectives; (6) clarify and discuss features of the architecture to support key performance scenarios; (7) analyze the architecture to determine whether it supports performance objectives; (8) identify alternatives, if problems are found; (9) present results; and (10) perform analysis of costs and benefits of the study and the resulting improvements.

As with other architecture evaluation methods, PASA starts with an overview of the method and architecture in steps 1 and 2. The critical use cases identified in step 3 are those that are important to the operation of the system, or that are important to responsiveness as seen by the user. They may also include those for which there is a significant performance risk. Each use case consists of a set of scenarios that describe the sequence of actions required to execute the use case. Step 4 focus on the scenarios that are executed frequently and on those that are critical to the user's perception of performance. Each key scenario has at least one associated performance objective. This objective is expressed in a quantitative and measurable way in step 5. The analysis of the architecture, performed in steps 6, 7 and 8, use several techniques, including the identification of the underlying architectural style and performance anti-patterns, and quantitative analysis through performance models. Steps 9 and 10 present the results and recommendation, which are complemented by an economic analysis showing that the time and effort dedicated to PASA was worthwhile compared to the time and effort that would be required if the problems were discovered later.

Despite being restricted to performance, PASA is interesting because it explicitly considers variations in the workload mix over time (specified as the conditions under which the required performance is to be achieved for each combination of scenario and objective). Nevertheless, the method also has weak guidelines for the specification of scenarios and performance objectives. In addition, despite the authors noting that performance objectives must be balanced with other quality concerns, they give no specific guidance on how to incorporate these conflicts and trade-offs into the analysis.

2.2.4 Performance Analysis and Prediction

Measurement and evaluation of certain qualities represented in our framework as dependent variables have been studied for many years. For this reason, our work is also related to analysis and prediction of specific system qualities, such as reliability, availability, security, and others. One area in particular that has received much attention is performance evaluation, where established models such as queuing networks, Petri nets and stochastic process algebras are used to estimate and compare the performance of one or more alternative system designs. These techniques are perfectly valid models to predict performance at different points of the system's operational range; but they generally fail to characterize the impact of the system's operational range on the *satisfaction* of its performance goals. One particular well known method in the area of performance evaluation is described below.

Software Performance Engineering (SPE)

Software Performance Engineering (SPE) is a systematic, quantitative approach to construct software systems that meet performance objectives (Smith and Williams, 2001). SPE is a model-based approach that uses deliberately simple models of software processing. Two types of model provide information for architecture assessment: the *software execution model* and the *system execution model*. The former is derived from UML models and is constructed using execution graphs to represent workload scenarios. Nodes in the graph symbolize functional components of the software, and arcs show the control flows. Solving this model provides a static analysis of the mean, best- and worst-case response times. If the software execution model indicates that there are no problems, analysis proceeds to construct and solve the system execution model. This model is a dynamic model that characterizes the software performance in the presence of factors that could cause contention for resources, such as other workloads or multiple users.

The SPE process includes the following steps: (1) assess performance risk; (2) identify critical use cases; (3) select key performance scenarios; (4) establish performance objectives; (5) construct performance models; (6) determine software resource requirements,; (7) add computer resource requirements; (8) evaluate the models; and (9) verify and validate the models.

In relation to the scalability framework, SPE suffers from the same weakness as PASA—weak guidance on elaborating scenarios, workload intensities and performance objectives.

There are many other works in the area of software performance analysis. Some of them propose extensions to standards and notations such as the Model Driven Architecture (MDA), the Unified Modeling Language (UML), the Architecture Description Language (ADL), the Ontology Web Language for Services (OWL-S) and the Web Service Offering Language (WSOL) to incorporate performance attributes (Woodside et al., 2005; Hopkins et al., 2002; Verdickt et al., 2004; Tribastone and Gilmore, 2008). Some works extend the SPE process (Hoeben, 2000; Bertolino and Mirandola, 2004). There are also works that rely on monitoring capabilities to model and predict performance (Mos and Murphy, 2004; Avritzer et al., 2002; Caporuscio et al., 2005; Avritzer et al., 2002; Reiss, 2008) and works that adopt an hybrid approach using modeling and monitoring (Liu and Gorton, 2004). These approaches can be specialized for different kinds of systems or technologies (e.g., component-based systems and middleware) by exploiting knowledge about the technology to aid performance prediction (Wu and Woodside, 2004; Llado; and Harrison, 2000; Petriu et al., 2000).

Although these works are restricted to performance, some of the models and techniques they propose can, theoretically, be used in the scalability framework to produce raw data for scalability analysis. Further research is required in order to integrate existing models for specific quality attributes into our scalability framework.

2.2.5 Multiple Criteria Optimization

A number of works in the computing literature deal with the problem of deciding among alternatives in the face of multiple, sometime conflicting objectives. These works can be found in a variety of fields, such as mobile computing (Capra et al., 2002; Shen et al., 2005), resource allocation (Lu and Bigham, 2006; Bennani and Menasce, 2005; Marbukh, 2007), autonomic computing (Cheng et al., 2006), testing (Lee and Snavely, 2007), recommendation systems (Wei et al., 2005), intelligent agents (Robu et al., 2005; Ficici and Pfeffer, 2008), Web services (Lamparter et al., 2007), and architecture evaluation (Kazman et al., 1998, 2001).

In this research, achieving scalability is seen as a multicriteria optimization problem, where the multiple, sometimes conflicting scalability goals have to be satisfied simultaneously. For such, we have devised a scalability analysis that shows to what extent these quality goals are satisfied when the characteristics of the application domain and system design scale. The stakeholders'quality goals are quantified by *preference functions* over the value of each quality attribute. A preference value therefore measures the "satisfaction level" of the stakeholder with respect to individual quality goals. A *utility function* transforms a vector of preference values into a single scalar value. The utility value can be thought as a measure of the "overall satisfaction" of the stakeholder with the scalability of the system.

From the works that use utility theory, the ones that most approximate the scalability framework are in the area of architecture evaluation. In ATAM, for example, a *utility tree* is used to translate from higher-level quality factors (e.g., performance, modifiability, availability and security) to concrete attribute scenarios (e.g., "transactions are secure 99.9% of the time"). The output of the utility tree provides a prioritized list of scenarios that is used to plan the amount of time spent by the development team in each scenario (Kazman et al., 1998). Therefore, the concept of utility in ATAM differs from that of the scalability framework, since it is just a mechanism for prioritizing scenarios.

In CBAM, utility is used to represent the benefit *added* to the system through the implementation of an architecture strategy (Kazman et al., 2001). A *utility-response curve* shows the utility as a function of the chosen response value. As described previously, this curve is built by asking the stakeholder to assign utilities to the worst-case, current, desired and best-case values of the quality attributes for a scenario, and scenarios are themselves assigned a normalized weight through a voting exercise. The benefit of an architectural style in a particular scenario is calculated by subtracting the utility of the expected value of the architectural style from the utility of the current system relative to this scenario. Finally, an *overall benefit* of an architectural style is the weighted sum of the benefits associated with each scenario.

A comparison can be made between CBAM and the scalability framework. CBAM's utility-response curve is similar to preference functions in the framework. Also, CBAM's overall benefit may be compared with the scalability framework's utility. However, there are fundamental differences:

1. In CBAM, the authors recognize that stakeholders have difficulty identifying the expected utility level of an architecture strategy. In the scalability framework, on the other hand, preference and utility functions are derived from a systematic requirements engineering technique.

2. CBAM is concerned with the cost/benefit trade-offs of changing the system architecture. Therefore, the concepts of *utility* and *overall benefit* refer to the *added* benefit brought by this change (i.e., *expected utility* minus the *current utility*). In the scalability framework, the *preferences* and *utility* refer to the range of quality attribute values in quality goals and the relative importance of these goals, independently of any change to the system architecture.

3. CBAM defines the weights in the *overall benefit* through a two-phase voting process. The scalability framework does not establish a particular method for deriving the utility function. Instead, a number of requirements prioritization techniques may be used for that purpose.

2.2.6 Scalability Requirements

As with other software analysis techniques, the correctness and usefulness of our scalability analysis is highly dependent on the selection of its variables and functions. However, it is all too easy to make these selections arbitrarily, as was noted by the authors of CBAM (Kazman et al., 2001). To address this problem, it is necessary to derive the variables, scaling ranges and functions in a clear, consistent and systematic manner from system requirements.

Many works in the scalability literature recognize the need to base the analysis on precise scalability requirements (Bahsoon and Emmerich, 2008; Williams and Smith, 2005; Weyuker and Avritzer, 2002; Smith and Williams, 2001; Steen et al., 1998; Masticola et al., 2005; Brataas and Hughes, 2004). However, the great majority of them take the elaboration of these requirements for granted. The few works in the area of scalability analysis that attempt to give some guidance on scalability requirements take an oversimplified view of the requirements engineering process. They also tend to concentrate on either specifying the information such requirements should contain, or providing rough guidelines on the scenarios such requirements should cover. Consider the examples below.

Steen et. al's Framework for Specifying Scalability Requirements

Steen et al. describe a framework for precisely formulating scalability requirements and an engineering method for building scalable distributed systems (Steen et al., 1998). The motivation for their work came from the observation that one requirement that is often formulated imprecisely is that an application should be "scalable".

The authors distinguish between two environments: a *client environment*, which consists of end users and processes that make use of services implemented by the application, and an *execution environment*, which models the infrastructure on which the application is executed. The client environment is associated with a number of attributes $Attr_1$, ..., $Attr_N$, such as the number of clients, number of requests, data volume, etc. This environment is also associated with a performance measure $Perf_A$, expressed in some numerical performance unit, such as throughput. The execution environment has a number of resources Res_1, ..., Res_M, whose combined capacity is associated with a cost. The authors also define a performance degradation bound δ (the maximum acceptable degradation in performance when increasing the value of the client environment attributes), and a reference cost bound γ (limiting the maximum costs for increasing resource capacity).

According to this work, in order to formulate scalability requirements precisely, one should define the attributes of the client environment, a simple performance measure, and the performance degradation bound δ. If resources are taken into account, one also needs the resources cost measure and the reference cost bound γ.

The main problem with this work is that it only addresses the *specification* of scalability requirements, taking the rest of the requirements elaboration process for granted. Elaborating scalability requirements is a complex activity, as will be discussed in Chapter 5.

PASA and QSEM

PASA and QSEM are scenario-based methods for performance and scalability evaluation of software architectures. As described in the Sections 2.2.2 and 2.2.3, these methods advise the analyst to choose scenarios that are executed frequently in the typical uses of the application, or that are executed infrequently but have high resource demand. They also advise on the specification of these requirements, which should combine the performance requirements, current workload volume, projected growth, and conditions under which the performance requirements should be achieved.

Although this work offers guidelines on the elaboration of scalability scenarios and objectives, these are still oversimplified. Without a systematic method, it is easy to overlook scenarios, particularly the ones triggered by exceptional circumstances.

Scalability in Requirements Engineering

To the best of our knowledge, there is no established work specifically on scalability in the requirements engineering field. Although the term scalability is often mentioned in the requirements engineering literature, it is normally in the context of the scalability of the proposed technique itself (In et al., 2001; Cheng and Atlee, 2007) or in very vague terms only (Bhushan and Patel, 1998; Heindl and Biffl, 2006).

In this thesis, scalability is seen as a meta-quality—a quality that relates to other system qualities. Therefore, a scalability analysis evaluates the degree of quality goal satisfaction when characteristics from the application domain and/or system design scale. For this reason we next consider a range of works on the elaboration of quality requirements; particularly the ones that allow for the evaluation of alternative designs with respect to degrees of goal satisfaction. This feature is interesting for two reasons. First because it would allow one to choose among strategies to achieve the desired levels of quality goals satisfaction, second because the degree of goal satisfaction may vary as characteristics of the application domain scale. Letier and Lamsweerde review a number of quantitative and qualitative reasoning techniques with this purpose. In summary, they conclude that:

> Qualitative approaches such as the NFR framework (Chung et al., 1999) and the Win-Win framework (In et al., 2001) are useful to gain a first, high-level understanding of the system quality goals and their interactions. However, for most systems, the information contained in such models is too vague to provide precise guidance for software architectural design, analysis and testing. For example, the NFR framework allows one to compute, for each alternative design the satisfaction status of each softgoal, namely "satisficed" (i.e. partially satisfied), "denied" or "undetermined" (Chung et al., 1999).
>
> Many of the quantitative techniques are based on quantitative values that have no precise physical interpretation in the application domain (Akao, 1990; Robinson, 1990; Yen and Tiao, 1997; Giorgini et al., 2002; Feather et al., 2002). For this reason, there is no way to test whether the level of satisfaction of quality goals are actually achieved in the running system. Other quantitative techniques are based on more precise specifications of quality goals expressed in terms of measurable criteria that have a precise physical interpretation in the application domain. The concept of "fit criteria" in the VOLERE (Robertson and Robertson, 1999) requirements engineering method is a typical example of this.
>
> Letier and van Lamsweerde (2004)

In this last category, is also Letier and Lamsweerde's work on partial goal satisfaction Letier and van Lamsweerde (2004). This work extends KAOS goal refinement graphs (van Lamsweerde, 2008) with a probabilistic layer for the precise specification of quality concerns expressed in terms of application-specific measures (Letier and van Lamsweerde, 2004). The technique is particularly interesting because it specifies partial goal satisfaction in terms of objective functions and quality variables, which can be used in downstream activities in the software development process.

Our scalability framework builds on Letier and Lamsweerde's work, not only to derive its functions and variables from KAOS goal refinement graphs, but also because some of the well-known advantages of goal-oriented requirements engineering are particularly useful for scalability analysis, as will be described in Chapter 5. More precisely, the scalability framework extends KAOS with the concepts of *scaling assumptions* and *scalability goals* and uses *goal-obstacle analysis* for reasoning about scalability during requirements engineering. The scalability framework also defines new patterns and heuristics for the *systematic* identification and resolution of *scalability obstacles*. This obstacle category had not been taken into account by previous techniques (van Lamsweerde and Letier, 2000). Furthermore, unlike previous patterns and heuristics that generate obstacles from one goal at a time, our technique consists of generating a scalability obstacle from *all* goals involving the same agent resource. This is an important and distinguishing feature of scalability obstacles. Thus, our technique allows one to identify a more precise, extensive and applicable set of obstacles than previously.

2.2.7 Capacity Planning

Another area of research that is closely related to our work is capacity planning. Capacity planning is the process of predicting when future load levels will saturate the system and of determining the most cost-effective way of delaying system saturation as much as possible (Menasce and Almeida, 2001). In fact, some of the techniques that are routinely used for capacity planning can be very useful for the scalability framework, such as regression analysis, analytical modeling and workload characterization. There are, however, fundamental differences between our work and capacity planning.

As described in Chapter 1, the scalability of a system is determined by its ability to satisfy quality goals when characteristics of the application domain and system design vary over expected ranges. Therefore, unlike capacity planning techniques that characterize the effect of scaling characteristics over measured system qualities, the scalability framework is also concerned with the effect of these characteristics on the *satisfaction of goals* concerning these qualities, as well as the relative importance that stakeholders give to the satisfaction of these goals. For this reason, the scalability framework also puts great emphasis on the elaboration of quality goals, which is generally taken for granted by capacity planning techniques.

Furthermore, the scalability framework makes a clear distinction between the scalability of a system (i.e., the effect of the scaling characteristics on the satisfaction of the system's quality goals) and the strategy used to achieve this scalability. It also treats cost as just another quality goal to be achieved by the system over the full scaling range of application domain and system design characteristics. Although the information produced by the scalability framework can be very valuable in defining a cost-effective way of delaying the system saturation, this is not its sole objective. The scalability framework and capacity planning techniques are, therefore, complementary approaches to the treatment of scalability.

2.3 Summary

In this chapter, we provided the theoretical grounding required to understand our scalability framework and gave an overview the state-of-the-art in scalability research and related fields.

Scalability has been studied in many contexts. Like ourselves, other authors have questioned the meaning and use of the term *scalability* and have attempted to provide better definitions. However, these attempts represent an intuitive ideal or are restricted to a narrow meaning. There are also a number of works concerned with analysing and predicting the scalability of software systems. The approaches for scalability analysis are varied. Some works, for example, attempt to adapt scalability notions from parallel computing to other classes of systems. There are also works that target narrow classes of systems and technologies, while still others are concerned with analysing the scalability of broader classes of software. However, in nearly all works we reviewed, scalability is analyzed with respect to performance metrics only.

Our work also relates to some architectural evaluation methods. This growing body of research is concerned with the analysis of software qualities at the architecture level. Depending on the method, the evaluation explicitly addresses a single quality or multiple quality goals at the same time. The latter may be perceived as overlapping with the scalability framework, particularly the ones that use multicriteria decision techniques. The main distinctions between these methods and our work have been discussed.

Finally, the scalability framework also relates to the field of requirements engineering, because variables and functions used in the scalability analysis are derived from system requirements. To the best of our knowledge, there is no established work specifically on scalability and requirements, and current NFR methods do not provide systematic guidance for elaborating scalability requirements. .

Chapter 3

Software Systems Scalability Defined

Our literature review suggests that scalability quotations normally refer to the characterization of the system response to the variation (or scaling) of some characteristics in the system's application domain and design. This chapter introduces a novel definition of the term that describes scalability in the context of other system qualities, and gives an overview a framework for the characterization and analysis of software system scalability.

3.1 On the Meaning of Scalability

In order to gain a first intuition on the subject of scalability, we resorted to extensive literature review. In particular, we looked for commonalities in scalability definitions and claims from both formal and informal literature (refer to Appendix A). Among them, a notable work on scalability was authored by Weinstock and Goodenough, the "On Systems Scalability" (Weinstock and Goodenough, 2006). They capture common intuitions on scalability in two definitions:

Def. 1. "Scalability is the ability to handle increased workload (without adding resources to a system)."

Def. 2. "Scalability is the ability to handle increased workload by repeatedly applying a cost-effective strategy for extending a system's capacity."

These are succinct and well-observed points on scalability. Yet, we can discuss a few drawbacks of the above definitions:

First, scalability is not always related to increase, but to *variation*. As an anecdotal evidence, take one of the earliest versions of the IEF. The various stages of the data analysis process were implemented as separate services, each running in its own Java Virtual Machine (JVM). At the beginning of the processing cycle, the JVMs were created, incurring a start-up delay that was virtually unnoticeable for the typical large batch of transactions. However, the IEF had to be scaled down to deal with small financial institutions—in that situation, the start-up delay represented a large chunk of the processing time of transactional batches. As a result, the user-perceived performance became unacceptable. The system, as implemented, could not be scaled down.

Second, another (yet imprecise) definition could be added to theirs:

Def. 3. "Scalability is the ability to handle the same workload more efficiently, by applying a cost-effective strategy for extending a system capacity."

In other words, a system can (possibly, repeatedly) be made scalable by realizing an improvement in its system qualities through the scaling of some characteristics of its design while maintained its environment unchanged. An illustration of such a definition of scalability can be found in the early days of the Web, when caching was created to improve the quality perceived by the end user, instead of anticipating the future growth of the Web (Fielding and Taylor, 2000).

Finally, as with other definitions, the authors associate scalability with the strategy to support the scaling of application domain characteristics and the costs for doing so. Such definitions essentially allude to how a designer achieves (stated or unstated) scalability goals, rather than referring to the effect the variation of some characteristics have on the satisfaction of the system goals. [1] We see cost as an important goal that has strong interaction with scalability and other goals. It is another goal that should be satisfied when devising or selecting among a set of scalability strategies. Bahsoon and Emmerich, for example, relate costs and scalability using mining in performance repositories to value the ranges in which a given software architecture can scale to support likely changes in the load (Bahsoon and Emmerich, 2008).

[1] In Chapter 5 the distinction between scalability goals and scalability strategies is shown explicitly.

3.1.1 Scalability, Variation and System Goals

Our literature review suggests that what scalability definitions have in common is the characterization of the system response to the *variation or scaling of some characteristics affecting its execution.* In fact, as in the example of the early days of the Web, scalability is not only about the system response to variation on its application domain, but also about the system's ability to satisfy varying quality goals for a static application domain. Therefore, variation may occur both in the *application domain* and the *system design*—that is, the world and the machine (Jackson, 2001)—and should be supported only *within expected ranges*. [2] The variation of application domain characteristics determines the *load* imposed on the system. The system's ability to support this load depends on its *capacity*, which may be achieved by varying the characteristics of the system design. Determining the critical scaling characteristics, the critical system qualities that must be measured in order to judge scalability, and what constitutes satisfactory values for these system qualities, are all things that *stakeholders ultimately must decide in the context of the system goals*, since they will know best what kinds of demands will be placed on a system and which system qualities are most critical to maintain or improve.

3.1.2 Scalability and Quality Goals

Scalability concerns have been related with different quality goals (as shown with added emphasis in the following quotes):

1. "Scalability is the ability of a system to continue to meet its *response time* or *throughput* objectives as the demand for the software functions increases." (Smith and Williams, 2001)
2. "However, a RAID5 system still has problems with its scalability ... the *MTTF* of RAID5 is inversely proportional to the square of the number of HDDs." (Yokota, 2000)
3. "The limited scalability of existing multicast simulation methods is primarily due to the large *amount of state* maintained by the simulators, which is often on a high order of the input size... This state requires a proportional *amount of memory* in the simulator." (Xu et al., 2003)
4. "The table above gives cost-equivalent key sizes ... The *time to break* is computed assuming that Wiener's machine can break a 56-bit DES key in 100 seconds, then scaling accordingly. The 'Machines' column shows *how many NFS sieve machines* can be purchased for $10 million..." (Silverman, 2000)

The use of the term scalability in the context of performance, as in the first quote, is the most common. However, performance is just one of many possible indicators of scalability. The second quote, for example, uses Mean Time To Failure (MTTF) to estimate the system availability as the number of hard disk drives scales. The third one refers to the amount of memory used by a simulation system as a function of its input size. And the fourth quotation points to a table that indicates the time required

[2] Weinstock and Goodenough also recognize the importance of specifying allowable ranges, stating that "no system is infinitely scalable" and "it is not reasonable to say a system is scalable without one or more demand measures with an allowable range of values and one or more response measures, with an allowable range of values".

to break symmetric keys using a varied number of NFS sieve machines; in this case, the performance metric “time” serves as a proxy for security.

Scalability cuts across system design decisions and refers to different system qualities, such as performance, reliability, availability, dependability and security. Scalability also relates to other quality attributes such as adaptability, extensibility and predictability. These attributes have a different relationship with scalability, as is discussed in Section 8.4. Scalability is therefore a meta-quality of other system qualities. That is, it is a quality that refers to the system ability to satisfy its other quality goals when characteristics of the application domain and system design vary over time. For this reason, it is vague to refer simply to “the scalability of a system”; instead one must refer to the scalability with respect to some specific measure of a system quality, such as “the scalability with respect to throughput”, or “the scalability with respect to latency and memory consumption”.

3.2 A Definition for Scalability

After extensive literature review and numerous discussions with researchers and practitioners, we settled on the following definition of scalability:

> **Definition:** *Scalability is the ability of a system to satisfy its quality goals to levels that are acceptable to its stakeholders when characteristics of the application domain (“the world”) and system design (“the machine”) vary over expected ranges.* (Duboc et al., 2008)

Scalability is not an absolute concept. Instead, it is always relative to a set of quality goals and how their level of satisfaction vary under the variations of characteristics in the application domain and the system design. In order to claim scalability, one must specify with respect to which quality goals, scaling characteristics and expected ranges the system is scalable. By “expected” we mean both normal and exceptional conditions (in which a degraded level of goal satisfaction might be acceptable).

This definition concisely addresses the issues highlighted in section 3.1, namely scalability being a matter of the system’s goals as determined by the stakeholders, and the system’s ability to support the variation of both the application domain and system design over expected ranges. Furthermore, this definition of scalability is independent of scaling strategies and their costs. Yet, cost is consider indirectly, as one of the quality goals to be achieved.

As a consequence of our definition, a *scalable system* is one that satisfies its quality goals when the characteristics of the application domain and system design vary over expected ranges. In order to demonstrate the expressiveness of our definition, consider the following example:

The Google architecture is a cluster of commodity processors over which the load of search queries is distributed (Barroso et al., 2003). In an article published in 2005, Barroso discusses two quality goals for the Google search engine: the ratios *performance/server price* and *performance/watt* (Barroso, 2005). The author looks back at three successive generations of Google server platforms, suggesting that Google is scalable because, over the years, it has managed to deliver increasing performance for roughly the same costs, resulting in an upward trend of the performance/server price. The article describes the

Google architecture at a very high level only and does not actually specify what is meant by performance. However, assuming that the number of queries per second and the number of indexed Web pages grew over the observed generations of Google, we can translate the author's claim as:

Claim 1: The Google search engine is *scalable with respect* to performance/server price (*quality of interest*) because it satisfies the goal of maintaining a upward trend in performance/server price (*cost goal*) as the number of queries per second and the number of Web pages to be indexed scale (*varying application domain characteristics*).

However, performance/watt has remained roughly flat over the Google generations observed by the author. In other words, every gain in performance has been accompanied by a proportional inflation in the overall platform power consumption. The author argues that if performance/watt was to remain constant over the years, it would lead to machines' powered up costs to be significantly more than the machines themselves, which he does not consider scalable. This claim would be translated as:

Claim 2: The Google search engine is *not scalable with respect* to performance/watt (*quality of interest*) because it does not satisfy the goal of achieving a less than linear performance/watt (*energy-efficiency and cost goal*) as the number of commodity servers scales (*varying design characteristic*).

Analyzing whether a system is scalable, or more scalable than an alternative solution, requires a careful characterization of the causal impact of the variation of application domain and design characterizes on system qualities of interest. The next section gives an overview of the scalability framework we developed with this objective. The framework is described in detail in Chapters 4 to 6.

3.3 The Scalability Framework: An Overview

Our scalability framework addresses the lack of a systematic and consistent treatment of scalability, allowing one to approach the problem of scalability analysis in a uniform manner across different system designs and application domains. Building on our definition of scalability, a framework for the characterization and analysis of software systems scalability requires a clear statement of the quality goals of the system, and the characteristics of the application domain and system design whose scaling will affect the satisfaction of these goals, as well as the feasible range of values for these scaling characteristics. This framework was first reported in (Duboc et al., 2007).

A high-level view of the elements composing the scalability framework is shown in Figure 3.1. These elements are represented by boxes. Larger boxes mean that more details will be filled in later. Arrows indicate the relationship between the framework elements.

As with any analysis, the framework starts with a **question** regarding the system's scalability for which an answer is being sought. Scalability questions must refer to particular **system qualities** (such as, *is Google scalable with respect to performance/server price when the number of concurrent queries and the size of the Web increase over time?*). The answer to a scalability question requires an understanding of the **quality goals** of the system (e.g., *an at-least-constant performance/server price*) and their required level of satisfaction when characteristics of the system's application domain (e.g., *the expected number of concurrent queries and Web pages in the foreseeable future*) and system design (e.g., *number of com-*

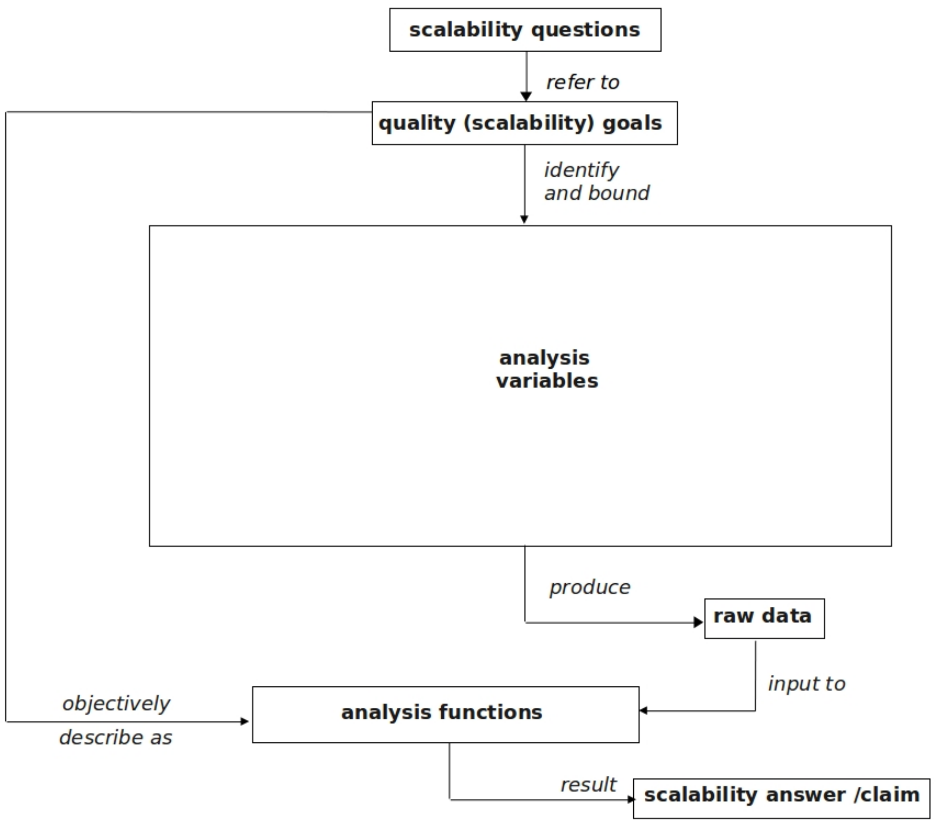


Figure 3.1: High-level view of the scalability framework elements.

modities machines in the cluster) vary over expected ranges. This information is objectively described as **variables** and **functions**. Testing and/or modeling produces the **raw data** for the quantitative analysis used to justify the **scalability answer/claim**. This high-level process is illustrated by Figure 3.2. Plain arrows represent data dependency, where the target step may require elements from the source step. For example, on one hand, knowledge of the system quality goals will inform what scalability questions need to be answered and, on the other hand, the question may inform which quality goals must be taken into consideration during the scalability analysis. Hence the bidirectional arrow in the diagram. Similarly, the result of a scalability analysis may lead to the revision of the system's quality goals.

The elements of the scalability framework are adapted from the main EPA steps (represented as dashed boxes) as in Figure 3.3: recognition of the problem, selection of variables and choice of experiment design, experimentation, analysis and interpretation, and conclusion.

3.4 Critical Evaluation

The risks associated with our definition of scalability and high-level view of the framework is as follows:

Risk: *Our definition challenges widely-spread intuitions on scalability.*

In particular, we observed that scalability goes beyond performance, that linear scalability is not always desirable, and that cost should be treated as another quality goal to be achieved when characteristics of the application domain and system design scale over expected ranges. We believe we have

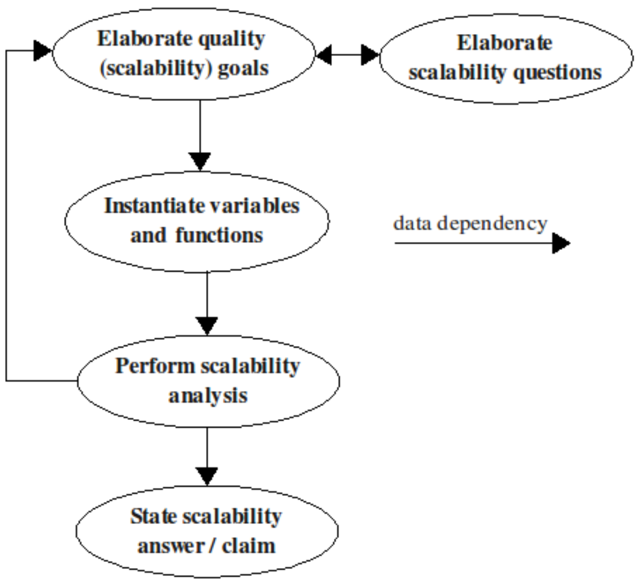

Elaborate quality (scalability) goals
Elaborate scalability questions
Instantiate variables and functions
data dependency
Perform scalability analysis
State scalability answer / claim

Figure 3.2: High-Level process for scalability analysis.

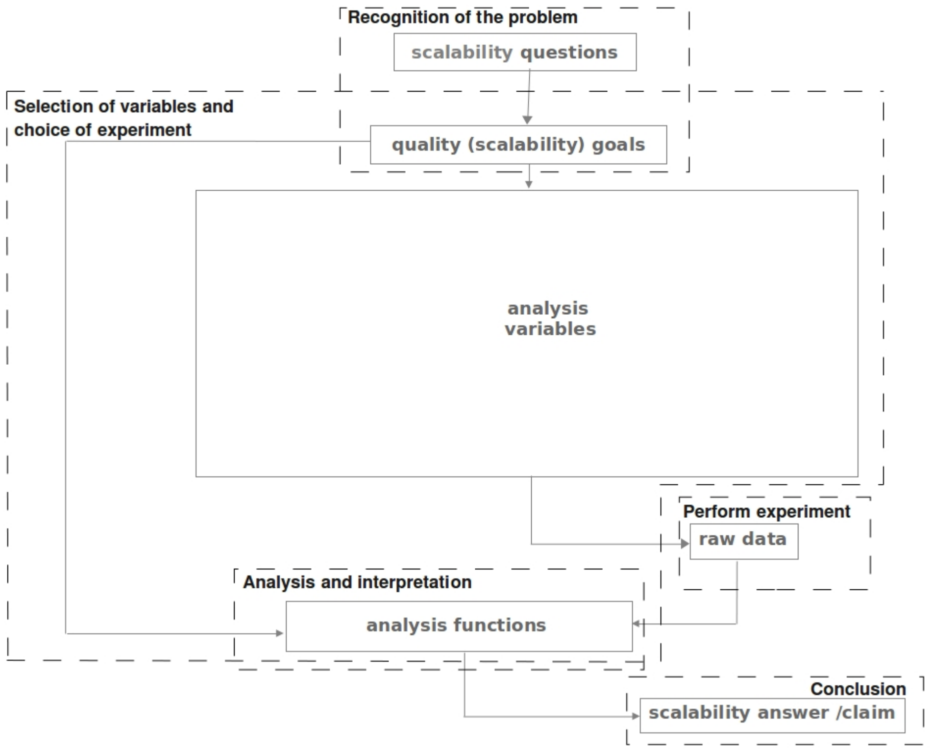

Recognition of the problem
scalability questions
Selection of variables and choice of experiment
quality (scalability) goals
analysis variables
Perform experiment
raw data
Analysis and interpretation
analysis functions
Conclusion
scalability answer /claim

Figure 3.3: Mapping of scalability framework elements to EPA.

constructed convincing arguments to support our statements.

Risk: *Attempting to create a general framework for scalability analysis that can be used across application domains and system designs could cause the perception of "stating the obvious".*

We identified essential elements of a scalability analysis that are common to all software systems (these will be explained in greater detail in Chapters 4 to 6). However, we provide no catalog of instantiations of analysis variables and functions for different categories of systems. Instead, we let them be instantiated according the system's scalability goals, which will differ greatly for one system to another. Given the broad application intended for the framework, it could not be different. Nevertheless, there is a risk that this generality would cause the perception of "stating the obvious". By no means do we think that is the case. Our extensive literature review and consultation with professional developers has convinced us that the term scalability is used with little precision and that scalability is not properly considered during the development lifecycle. Undeniably, outstanding developers have been considering, analyzing and achieving scalability for years. However, the larger software community is still lacking a uniform, consistent and systematic treatment of scalability. Our framework tackles this problem by providing *systematic steps* for instantiating the elements of a scalability analysis, allowing to identify and resolve scalability problems that could be overlooked otherwise.

Risk: *Developers may feel that the costs of applying the scalability framework outweighs its benefits.*

Undoubtedly, regardless of the numerous benefits that such analysis is likely to bring in terms of better defined requirements, higher design quality, and longer lifetime for the system design, an upfront analysis of scalability will require time and effort. We have observed in our interviews with practitioners that scalability problems often become apparent only when the system is exposed to full load during the production phase (Industry Interviews, 2007). This is also the phase where problems are costliest to fix (Boehm, 1976), making the analysis of scalability at earlier stages an attractive alternative. However, further research is needed to assess the portion of development time the adoption of the framework would take and how it would fit into the popular software development methodologies.

Contributions and Benefits

Our definition and framework were deliberately designed to convey different notions of scalability and to be used across a wide range of software systems. For such, we created:

Contribution: *A uniform and precise definition of scalability that is independent of application domains and system design.*

Unlike definitions that associate scalability with particular metrics, scaling characteristics or scaling strategies, ours describes scalability in terms of quality goals and scaling characteristics of the application domain and system design. These are concepts common to all software systems, resulting in a uniform and precise definition that transcends application domains and system design.

Further benefits associated with the ideas discussed so far are as follows:

Benefit: *Counter-arguments to common misconceptions on scalability.*

In our conversations with practitioners and other researchers, and in our extensive literature review, we observed a number of misconceptions with respect to scalability. Possibly the two most common ones are the belief that scalability is always related to performance and that a scalable system must present an at most linear increase in resource usage as demands on the system increase.

Benefit: *The recognition of scalability as a quality that can be analyzed with respect to other system qualities.*

The misconception that scalability is always related to performance is a dangerous one, as it may lead the developer to neglect the effects of scaling system characteristics on other system qualities. In this work, we explicitly state that scalability should be analyzed with respect to quality goals. Therefore, when describing the scalability of a system, one should clearly describe it with respect to which specific measures of system qualities the system is scalable, such as "scalable with respect to throughput", or "not scalable with respect to latency and memory consumption".

3.5 Summary

In this chapter, we have defined scalability as *the ability of a system to satisfy its quality goals to levels that are acceptable to its stakeholders when characteristics of the application domain ("the world") and system design ("the machine") vary over expected ranges*. In this definition, scalability is seen in the context of the system's quality goals, so it can relate to performance, availability, reliability, security, costs and so forth. We have also presented a framework, based on EPA, for the characterization and analysis of software system scalability. The framework is composed of six elements: the scalability question, the scalability goals, the variables and functions to be used in the scalability analysis, the raw data and the scalability answer or claim.

The next chapter elaborates on the framework, presenting the details of the analysis method and explaining how it can be used to evaluate and compare the scalability of software systems.

Chapter 4

Characterizing and Analyzing Scalability

This chapter drills into the details of the scalability analysis technique. It discusses common causes of scalability problems, explains the semantics of variables and functions in the scalability analysis, and illustrates how the analysis can answer different scalability concerns.

4.1 On Scalability Analysis

We have seen from our conversations with developers in industry that many did not properly consider scalability when building systems. They frequently made unfounded assumptions about or disregarded the effect that the chosen design had on the satisfaction of the system's quality goals when characteristics of the application domain scale or its ability to satisfy varying quality goals. A common assumption is that the system would gain scalability with more powerful hardware or parallelization (Industry Interviews, 2007). Nevertheless, achieving scalability through such strategies is not straightforward. In fact, without a careful analysis, they can provoke just the opposite result due to knock-on effects, such as the overhead of distributed communication or race conditions in threads (Noelle et al., 1998; Yu, 2008). Also, the scalability achieved by varying the system's hardware is limited by physical constraints, such as the speed of light limiting the latency of network packet delivery and a "thermal wall" bounding the clock speed (The 2020 Science Group, 2006). In addition, some of the developers who claimed to have considered scalability had catered for the extreme cases of the application domain only (Industry Interviews, 2007)—considered by us to be a fallacy, as discussed in Chapter 1.

Analyzing scalability requires, instead, a clear and precise picture of the operational impact that the *variation* of application domain and system design characteristics will have on system qualities. In an ideal world, that picture would cover the *full operational range* of multiple scaling characteristics. However, in reality, the complete picture will rarely be feasible or, in fact, necessary. Instead, this picture is required to cover *a sample of points* of the operational range of these characteristics. It is the stakeholder who must determine which are the relevant points and the desired level of system qualities at these points, according to the system's scalability goals. For example, the stakeholder may choose the sub-range of values that generate most of the company's revenue plus the extreme points of the scaling ranges. Such understanding is useful for guiding design decisions that are more likely to support multiple deployments, to state scalability claims objectively and to compare claims from different sources. We, therefore, define scalability analysis as:

> *Scalability analysis is a form of experimental design that is used to reveal—in a precise and explicit form—the impact caused by the relevant points in the operational range of the system application domain and design characteristics on the satisfaction of the system's quality goals.*

Consequently, *any system analysis—performance, reliability, availability or other—conducted with respect to a variation over a range of application domain or system design characteristics is a scalability analysis.*

4.2 Causes of Scalability Problems

The scalability of a system is influenced by a number of factors, including its operational environment, the underlying operating system, network, database and programming language (Munsterman, 2008), and it can affect a number of system qualities. Weinstock and Goodenough observe that scalability failures occur when increased demand causes some resource to become overloaded or exhausted, such as exceeding the available address space, overloading the memory, exceeding the available network bandwidth, and filling the internal tables (Weinstock and Goodenough, 2006). Nevertheless, increase in demand and resource exhaustion are not the only reasons for scalability problems. The authors themselves observe that a scalability problem may be caused by a bug, as in the case of a memory leak. Another example is given by the IEF, whose scaling down—in number of bank transactions to be processed by the system—increased the user-perceived response time due to the start-up overhead of JVM's. We therefore state that a *scalability problem occurs when the scaling of characteristics of the system's application domain and system design prevents the system's quality goals from being satisfied.*

Sometimes, scalability is completely overlooked during system development. On other occasions, scalability is considered, but problems occur because the system relies on assumptions that do not hold true when its application domain and/or system design scale. These are assumptions such as the correctness of configuration files, the system's usage pattern, advances in technologies, and the rate and effect of "rare" failures or exceptions (Weinstock and Goodenough, 2006). Take, for example, an exception that increases the load on the system (e.g., by writing error messages to a log file). The increase in load of such an exception may not be considered a problem, if its occurrence is believed to be rare. However, the scaling of some domain quantity may affect the number of occurrences of this exception, which can, in turn, lead to the exhaustion of some system resource (e.g., disk space).

Other causes of scalability problems include changing requirements, "hard-coded" capacity limits, changes in the usage pattern caused by advances in technology or acquired knowledge of the system, knock-on effects of changes to the system's implementation or configuration, unplanned evolution (in particular, from bespoken projects to a single product), exceptional application domain circumstances, inadequate design, tools or infrastructure, lack of knowledge of the adopted technologies, and inexperienced developers (Weinstock and Goodenough, 2006; Industry Interviews, 2007). A summary of causes for scalability problems observed during our industry interviews is given in Appendix B. We now illustrate scalability problems caused by some of these scenarios:

"Hard-coded" capacity limits: "Comair is the commuter affiliate of Delta Airlines. During the Christmas travel season in 2004, Comair stranded thousands of airline passengers because its 10-year-old crew scheduling system crashed. The reason for the crash was that the system was only capable of accommodating 32,000 (probably 32,767) crew schedule changes in a month. This limit was by design and was presumably acceptable—until it wasn't." (Weinstock and Goodenough, 2006)

Knock-on effects from changes: As Searchspace, the company who developed the IEF, expanded to new markets, the system had to be adapted to support additional languages and regional differences. This adaptation demanded a change from single-byte to multi-byte characters. Consequently, there was an increase in the size of the data to be manipulated by the system, forcing Searchspace to re-evaluate the use of resources, such as memory and disk, to prevent a scalability problem.

Exceptional circumstances: A real-world system, used by a large transportation company, aggregates data from various external data sources, making them available to a number of internal applications. The system also performs a number of heavy batch management tasks. The system was designed such that batch files were scheduled not to coincide with each other. However, occasionally, external data sources would become unavailable, forcing the system to wait and causing the overlap of batches. This overlap would lead to an increase in the data volume and the exhaustion of the system's resource (Industry Interviews, 2007).

Unusual system usage: A Web-based booking system for a large theater offers users the ability to see the view of the stage from the chosen seat. Users, however, desiring to check the view of multiple seats in order to choose one, would open a number of tabs in the Web browser—one for each potential seat. On the occasion of a popular play, this unpredicted system usage caused a large increase in the number of connections to the Web server, bringing the system to a halt.[1]

4.3 Elements of the Scalability Framework

In Chapter 3, we have introduced a framework for the characterization and analysis of software systems scalability. The elements composing this framework are quality (scalability) goals, scalability questions, analysis variables, analysis functions, raw data, and scalability answer/claim. A more detailed view of the framework is given in Figure 4.1. Note that analysis variables have been divided into *factors* and *dependent variables*. The former represent characteristics of the application domain and system design that affect the system behavior, and the latter are metrics that characterize system qualities of interest in a scalability analysis. This analysis, however, is not performed with respect to the values assumed by the system qualities, but according to the stakeholders' satisfaction with these values and the relative importance they give to the system quality goals—respectively described in the framework by *preference* and *utility functions*. This analysis has been published in (Duboc et al., 2007).

We next describe the elements composing the scalability framework. Emphasis is given to the analysis variables and functions, which are the building blocks of the scalability analysis technique discussed in this chapter. Other elements will be revisited later in the thesis.

4.3.1 Quality (Scalability) Goals

We define a *scalability goal* as a goal that specifies the acceptable levels of satisfaction for *quality goals* with respect to the scaling of application domain characteristics. A careful elaboration of such goals is required both for asking relevant scalability questions and for deriving the variables and functions to be used in the scalability analysis. The elaboration of scalability goals and their use in the framework are described in Chapters 5 and 6.

4.3.2 Scalability Questions and Answers/Claims

The scalability question defines what information should be uncovered by the scalability analysis. Analyzing a system's scalability is a complex, multi-variate activity and no analysis can realistically take

[1] Adapted from a real story.

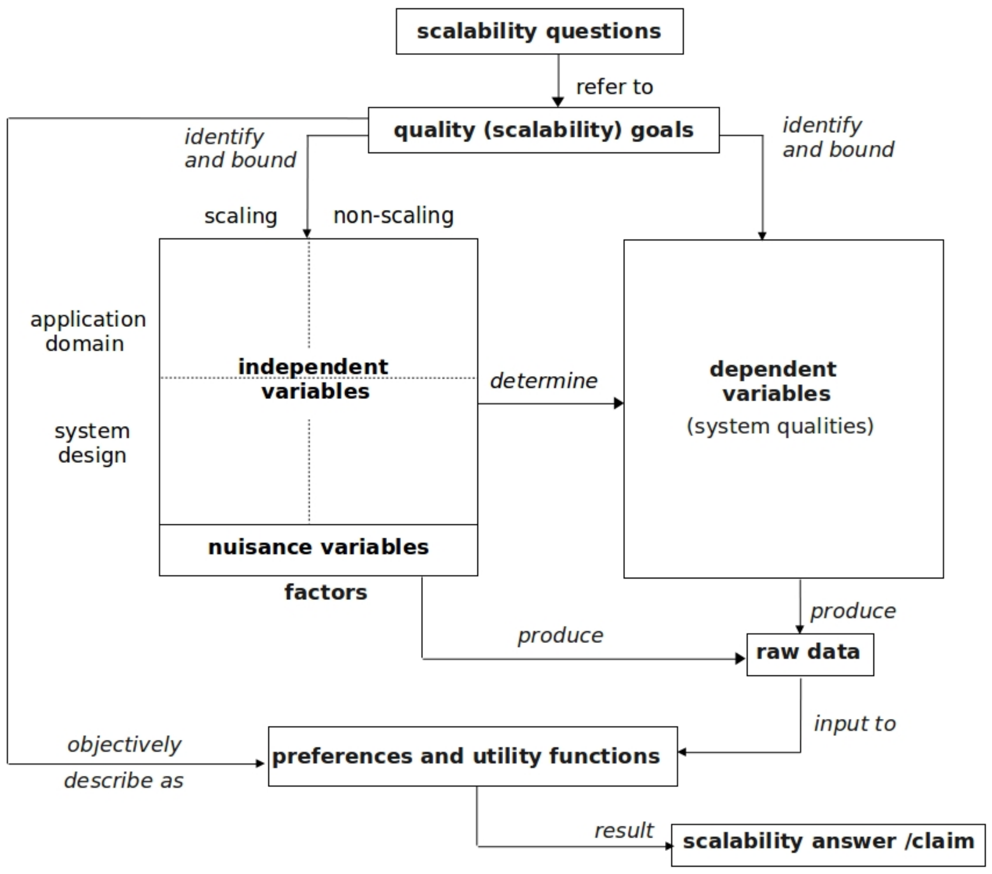


Figure 4.1: Elements of the scalability framework.

into consideration all quality goals, application domain characteristics, and strategies to achieve scalability at once. A scalability question, therefore, should focus on a particular scalability aspect of interest. Normally, analyzing the scalability of a system will involve a number of scalability questions.

A scalability question is often related to one of the two concerns: checking whether the system can satisfy its quality goals when characteristics of its application domain scale, and exploring the system's limits and scaling trends. Questions may refer to a system in isolation, or compare systems (or alternative system designs) with respect to the above concerns. Much of the information required for stating a scalability question can be obtained from the system's scalability goals, as will be described in Chapter 6.

The exact format of the question is highly dependent on the concern being addressed and on the information available at the time. Take for example, the scalability question introduced with the Google example in Section 3.3. In its simplest form, a scalability question should refer to a system quality and one or more application domain characteristics whose scaling is likely to affect the value of this quality:

Question Is Google scalable with respect to performance/server price when the number of concurrent queries and the size of the Web increase over time?

Later, the question may be refined to refer to a particular scalability goal and to include characteristics of the system design whose scaling are likely to affect the system qualities of interest, as follows:

Question Can Google maintain an at least constant trend in performance/server price when the number of queries per second and the number of Web pages to be indexed scale by increasing the number of commodity machines in the Google cluster?[2]

Similarly, a scalability answer or claim should be described clearly and with as much detail as possible. An example of a scalability answer/claim has been given in Section 3.2. In this example, a claim for Google's scalability was stated as follows:

Claim: The Google search engine is scalable with respect to performance/server price because it satisfies the goal of maintaining a upward trend in performance/server price as the number of queries per second and the number of Web pages to be indexed scale.[3]

4.3.3 Analysis Variables

Factors represent characteristics of the application domain and system design that will affect the system behavior, such as the volume of input data, the rate at which work arrives at the system, the number of concurrent system users, the maximum cache size, the maximum thread pool size, the number of nodes in a server cluster, and the algorithm selection. These characteristics can be classified as scaling and non-scaling.

The subset of the factors that can be manipulated for the scalability analysis are called *independent variables*. *Scaling independent variables* represent characteristics of both the application domain and the system design that can potentially *scale*, whether within or in between executions of the system. They could be, for example, the number of concurrent users or available nodes in a cluster. The *non-scaling independent variables* are characteristics of the application domain or system design that are either set to fixed levels or vary within a nominal scale. They could be, for example, the selection of a sorting algorithm or the RAM of the machine. *Nuisance variables* are characteristics of the application domain and system design that cannot be manipulated for the scalability experiment, such as the processor speed if the infrastructure is already decided and cannot be changed; these characteristics may also scale.

Dependent variables are metrics that characterize system qualities such as performance, maintenance, reliability, and security. Examples of dependent variables are peak and average values of throughput, storage consumption, and failures per unit of time. While the dependent variables typically represent metrics of traditional quality goals, it is the analysis of the dependent variables *in the presence of variation to the scaling variables* that turns an ordinary quality analysis into a scalability analysis.

The selection of factors and dependent variables are unique to the scalability question and system at hand. Once the variables of interest have been identified, the stakeholder must establish bounds defining their expected range of values. The bounds reflect the expected variation in the application domain, the technical and economic feasibility of the system design, and the acceptable bounds on required system qualities. Note that the system may be designed to cope with variation in scaling variables by

[2]Question adapted from (Barroso, 2005). Scaling characteristics and their possible range of values have not been described in that paper.

[3]Answer adapted from (Barroso, 2005). Possible range of values for the number of queries/second and Web pages have not been described in that paper.

dynamically varying some characteristic of the design, such as the size of a thread pool. However, such dynamic variation can be performed only within static limits, and it is these static limits that are the bounds for the independent variables representing the system design. When possible, the stakeholder should also determine the probability distribution of the independent variables. Section 4.4 discusses the usefulness of this information on a scalability analysis.

As an example, take the Google search engine: In 2003, Barroso et al. described the two most important goals that influenced the Google design: the energy-efficiency and price-performance ratios. These goals are achieved through a large cluster of commodity processors over which the load of search queries is distributed (Barroso et al., 2003; Barroso, 2005).[4] In this Google example, the elements of the scalability framework are instantiated as follows:

Quality Goals:

Cost goal: *maximize performance per unit of cost*

Energy-efficiency goal: *minimize power usage per server*

Independent variables:

Scaling Variables:

Number of queries per second (application domain)

Number of Web pages to be indexed (application domain)

Number of machines in the Google cluster (system design)

Non-scaling variables:

Available bandwidth (system design)

Network round-trip time (application domain)

Dependent variables:

Aggregate request throughput (performance)[5]

Performance/server price [6]

Performance/watt [7]

4.3.4 Analysis Functions

We see achieving scalability or choosing among alternative systems with respect to scalability as a multicriteria optimization problem, in which several, possibly conflicting quantitative criteria are to be optimized simultaneously. For this reason, our scalability analysis shows to what extent these quantitative criteria are being met when characteristics of the application domain and system design vary over time. In the scalability framework, the dependent variables quantify the criteria to be optimized, and the trade-offs among the dependent variables result in different choices of system designs.

[4]The instantiation of the framework variables is limited by the information available in the papers, which do not describe the ranges or specific targets for quality goals.

[5]It is not clear from (Barroso, 2005) which performance metric is used. From (Barroso et al., 2003), we assumed it to be aggregate request throughput.

[6]Performance per sum of capital expense (with depreciation) and operating costs, such as hosting, system administration and repair (Barroso, 2005).

[7]Performance per overall platform power consumption (Barroso, 2005).

Preferences

The stakeholders' scalability goals with respect to different measurable characteristics of the system are quantified by *preference functions* over the values of the dependent variables. The analyst can then use these preference functions to measure the "satisfaction level" of the stakeholder with respect to individual quality goals. The preference functions must be defined in such a way that the qualities of the system are mapped to a range of values in which a greater value equates with greater preference. For instance, the identity function could be used as a preference function for a quality such as average throughput, since higher throughput is normally preferred to lower throughput. However, for a quality such as memory consumption, lower consumption is normally preferred to higher consumption, and so the preference could be computed as the reciprocal of the value of memory consumption.

In many multicriteria optimization problems, a solution that maximizes all preferences may not exist. For instance, it may be possible to improve the throughput of a system at the cost of memory consumption due to increased buffer size, or conversely to reduce memory consumption at the cost of decreased throughput. In such Pareto optimal situations, it is not possible to improve any one quality without compromising another (Varian, 2003). Typically, the Pareto optimum will be a frontier of points whose shape indicates the trade-off between different objectives. In our framework, preferences over the dependent variables of interest define the axes of the plot of the frontier. Different system designs will produce alternative outcomes (i.e., points in the plot), which may or may not lie on the Pareto frontier.

Utility Functions

The choice between alternative outcomes in a Pareto frontier is not straightforward. To resolve the trade-offs inherent in a Pareto optimal situation, we use a *utility function* to transform a vector of preference values into a single scalar value. The utility value can be thought as a measure of the overall satisfaction of the stakeholder with the system.

A commonly used model of utility is *objective weighting*, in which the utility function $U(x)$, for a random variable X, is a linear combination of preference values $f_j(x)$, with each preference value weighted by a corresponding weighting factor λ_j (Eschenauer et al., 1990). Therefore, taking X to represent the values assumed by a scaling independent variable and $f_j(x)$ to be a preference over dependent variable j, the utility $U(x)$ is calculated as $U(x) = \lambda_1 f_1(x) + \ldots + \lambda_k f_k(x)$ for k dependent variables. The weights represent the extent to which an increase of one quality will be accepted as the others decrease. The choice of weights reflect the interests of the stakeholders, who must decide how to prioritize the preferences over particular system qualities.

When assigning weights to disparate system qualities such as disk consumption and throughput, there is a danger of producing an ill-defined utility function due to wide variation in the range of possible values for each preference. Therefore, the preference values $f_j(x)$ should first be *normalized* to a common range before being used in a utility function. For instance, we may define a function $\hat{f}_j(x)$ for each dependent variable j that normalizes the preference values $f_j(x)$ to a value between 0 and 1:

$$\hat{f}_j(x) = \frac{f_j(x) - f_{j,min}}{f_{j,max} - f_{j,min}} \tag{4.1}$$

where $f_{j,min}$ and $f_{j,max}$ are, respectively, the lowest and highest possible preference values for the dependent variable j. If the lowest and highest possible values cannot be established in a meaningful way, then the lowest and highest preference values for all observed outcomes can be used.

4.3.5 Raw Data

We refer to *raw data* as the data used to calculate the value of preferences and utility in the scalability analysis. In other words, raw data are all the values assumed by the dependent variables in a scalability analysis. Raw data may be generated in a number of ways, such as testing, modeling or simulation. For example, one could build a prototype of the system (or part of it) with which to run experiments. Alternatively, the analyst could create a queuing network model to estimate the system performance at different points of the scaling ranges of independent variables. In this research, we only employed testing, using both a real system and prototypes of it. [8]

When using testing, the values of the dependent variables represent an observation of the execution of a single system design. An observation may be a snapshot of the system at any given time or may be some measurement of interest (average or peak) of a complete system execution.

4.4 Scalability Analysis

A simple scalability analysis can be made by plotting the utility outcomes against the scaling variables. Consider, for example, a scalability analysis comparing five different hypothetical system designs with respect to the dependent variables average response time and peak disk usage. The axes of the two-dimensional graph in Figure 4.2.a represent the preferences over these two dependent variables. This figure, therefore, shows a snapshot of these preferences produced by the alternative designs for a particular value of the scaling independent variable average number of simultaneous users. For this snapshot, the outcomes produced by designs A, B and C are Pareto optimal. In particular, none of these three designs is universally preferable to the other two, since none of them maximizes both preferences. However, designs D and E are inferior in both preferences for A, B and C. If that is the case for the full range of the scaling variable, designs D and E can be rejected from further consideration. Note that a system design may under perform for particular values of the scaling variables, but still represent an acceptable solution for the majority of the values in the scaling range.

Figure 4.2.b plots the utility curves for the three Pareto-optimal designs of Figure 4.2.a against the average number of simultaneous users. We have used as a utility function a weighted sum of the normalized preference values over the dependent variables, where the preference for peak disk usage receives three times the weight as preference for average response time. The design that maximizes utility for the expected range of the scaling variables is the one that should be selected. In this example, the design C quickly overcome the utility of the other designs at around 30,000 users. Whether the design C should then be chosen will depend on the likely range and distribution of the number of simultaneous users.

[8]The implications of this decision are discussed in Section 4.5.

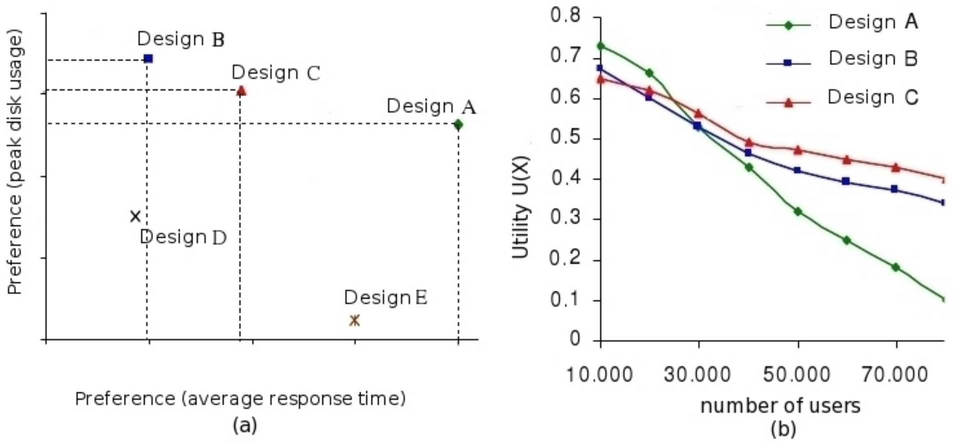


Figure 4.2: Comparing alternative designs.

Probability Distribution of Independent Variables

A particularly desirable information for a scalability analysis is the expected distribution of the scaling variables. Consider, for example, the graph in Figure 4.3, where A and B represent alternative system designs whose utility is plotted against the scaling variable *number of business entities* (x). Note that there is a critical point c', from which the utility of design A overcomes the utility of design B as the number of entities increases. If we consider that the graph covers the full range of the scaling independent variable (i.e., no extrapolation is necessary) and that the preference and utility functions were designed in such a way that a utility of zero indicates non-compliance with quality goals, we can conclude that both designs are feasible since none of them reaches zero utility at any point of the graph. Nevertheless, the choice between designs A and B will depend on whether the likely number of entities the system is required to handle lies before of after the critical point c'. If, for example, the distribution of the number of entities is as depicted by the probability density function (pdf) $p(x)$ shown in Figure 4.4, then the probability $P(X \leq c')$ that the system will have to handle less than c' business entities is high, approximately 0.9. If, however, the pdf $p(x)$ is as described in Figure 4.5, then the probability $P(X \leq c')$ that the system will have to handle less than c' business entities is low, approximately 0.3. Clearly, in the former case, design B should be chosen, while in the later one, design A is more desirable. Mathematically, the utility of a system design for a number of business entities described by $p(x)$ is calculated by Formula 4.2, where X is distributed in the interval $[C_{min}, C_{max}]$. [9]

$$U = \int_{C_{min}}^{C_{max}} U(x)p(x)\,dx \tag{4.2}$$

[9]Modeling multiple scaling characteristics requires accounting for any dependency between their corresponded variables.

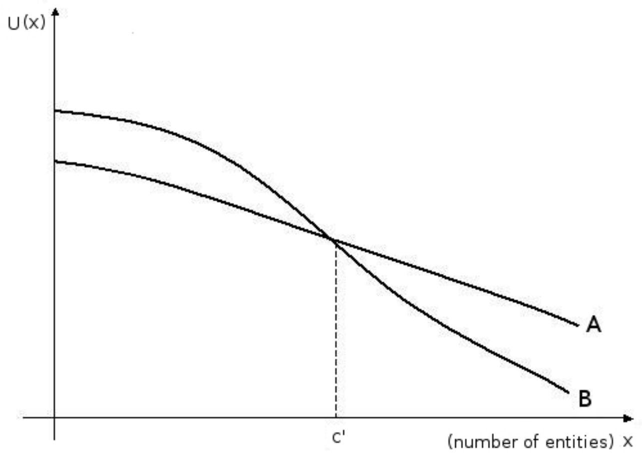


Figure 4.3: Utilities of two hypothetical alternative designs against the number of business entities.

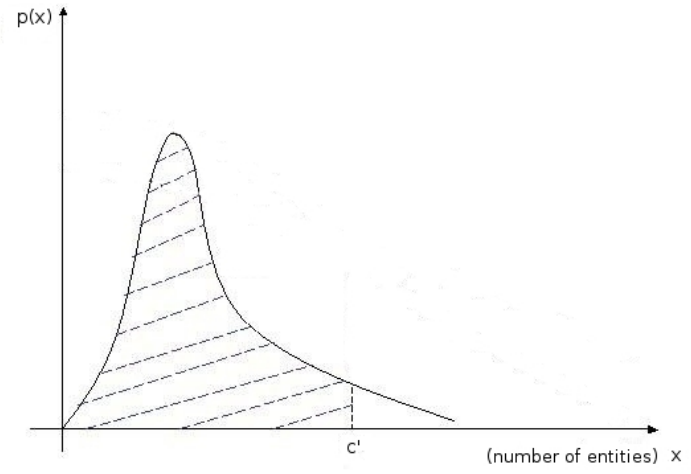


Figure 4.4: Hypothetical pdf for the number of business entities.

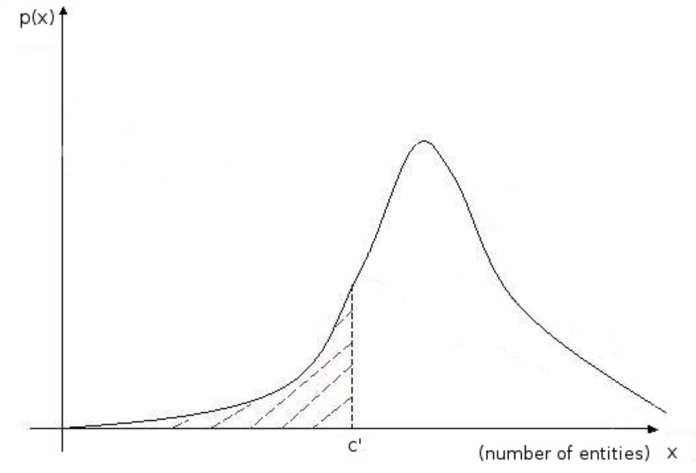


Figure 4.5: A different hypothetical pdf for the number of business entities.

4.5 Critical Evaluation

The risks associated with the scalability analysis described in this chapter are as follows:

Risk: *The usefulness of the analysis results depends on the adequate identification of application domain/design characteristics, system qualities, and functions.*

A performance consultant, with whom we discussed our framework, noted that normally scalability problems arise from overlooked variables (Sorensen, 2007). His opinion was confirmed by the industry interviews we conducted throughout this research. Scalability problems indeed often took developers by surprise. These problems were the result of overlooked characteristics, wrong assumptions, unexpected circumstances and knock-on effects of changes to the software, as summarized in Appendix B. In order to mitigate this risk, we introduce in Chapter 5 a technique for systematically selecting variables and functions for a scalability analysis. This technique includes obstacle analysis to help identify exceptional conditions that may prevent the systems' scalability goals from being satisfied.

Risk: *Relying on software metrics and optimizations for characterizing and analyzing scalability may be misleading.*

This was, in fact, a true criticism received by this work in the early days of the research (Alspaugh, 2007). The main points of this particular observer were:

- The framework requires a "leap of faith" in the relationship between the metrics that are collected and system qualities we would like to measure.
- Metrics are very sensitive to the way they are collected, so one cannot really trust the metrics being collected or what is inferred from them.
- Optimization and decision functions are, in general, very sensitive to small errors in measurement, abnormalities and wrong assumptions. In addition, choosing a function would require a unreasonable amount of work in analyzing that function and its sensitivity to the various factors, the properties and correlation of the variables, etc.—which is simply not practical.

Another understandable concern refers to the adequacy of such functions to compare alternative systems/designs. Alternatives may represent radically different designs, and care must be taken in defining the semantics of variables and functions so that they can be used for comparison.

These are known problems with metrics and optimization/decision functions in the general case, however we do not think they are strong enough reasons to disregard the scalability framework. We use the words of Fenton and Pfleeger to respond to these concerns:

> *"It is the general picture that is important, rather than the exactness of the individual measure, so the subjectivity, although a drawback, does not prevent us from gathering useful information about the entity"*—Fenton and Pfleeger (1996)

> *"Those who reject a measure because it does not provide enough information may be expecting too much of a single measure. [...] Likewise, models that define an attribute in terms*

of several internal attributes may be useful, even if they are not complete."—Fenton and Pfleeger (1996)

Finally, another fair criticism of the analysis method would be its use of a weighted sum. Direct estimation of weights is subjective and vulnerable to judgment errors. A similar observation was made by Letier and Lamsweerde with respect to qualitative requirements engineering methods that use weights for determining the degree of goal satisfaction (Letier and van Lamsweerde, 2004). Instead, Letier and Lamsweerde suggest integrating quantitative techniques for reasoning about non-functional properties with goal-oriented techniques for building requirements models. In Chapter 5, we build on their solution and suggest the use of requirements prioritization techniques to reduce the risk of deriving inappropriate variables and functions for the scalability analysis.

Risk: *Input distributions and growth characteristics of independent variables may not be known.*

A scalability analysis can benefit greatly from knowledge of input distributions and growth characteristics of independent variables. Such characteristics can be inferred from field usage data, also known as an *operational distribution* or *operational profile* (Musa, 1993). However, collection of field data can be both expensive and time-consuming, and in some cases, it may even be impossible (Weyuker and Avritzer, 2002). Clearly, if the distribution of independent variables cannot be predicted reliably, an analysis simply considering different points of the scaling range can already bring useful information (for example, it can indicate whether the system can meet its goals for all the points considered). However, when possible, a careful workload characterization should precede the scalability analysis. The amount of effort worth putting into this activity is highly dependent on the problem and system at hand. Weyuker and Avritzer, for example, believe that "in spite of the difficulty and expense of collecting data in the field, it is definitely worth the resources when it is essential to have highly dependable systems" (Weyuker and Avritzer, 2002).

Risk: *It is impossible to test realistically for scalability.*

Weinstock and Goodenough observe that: "You can't really test (in the traditional sense) to see if a system is scalable. It is usually impossible to generate a realistic load—much less to actually add computing capacity to see if the planned scaling strategy will work smoothly" (Weinstock and Goodenough, 2006). When the the full workload cannot be generated, and the infrastructure is not powerful enough to test the system (or a prototype), then extrapolation is necessary. Due to time constraints, however, the extrapolation of the analysis results had to be left out of the scope of this PhD research. Therefore, for the context of this work, we assumed:

Assumption: There ultimately will be a running version of the system, possibly a prototype, that can be used for scalability analysis.

Assumption: There are sufficient data and hardware infrastructure to run the system (or a prototype) over the full scaling range of independent variables for purposes of analysis.

We recognize these are strong assumptions that will not always hold in the real world. Yet, they do not invalidate the contribution of this work, namely a precise and uniform definition of scalability and a framework that is independent of application domain and system design. Furthermore, nothing in the framework prevents raw data from being generated through modeling or simulation, or the framework from being further elaborated to include techniques for the extrapolation of analysis results. This is, in fact, one of our directions for future work. Extrapolation of the analysis results is difficult for a number of reasons. It is not straightforward to extrapolate the combination of characteristics with different scaling trends. In fact, even extrapolation of individual variables should be informed by knowledge of the underlying computational phenomena determining the scaling trend of that variable. Extrapolation of the analysis results is discussed in more detail in Chapter 8.

Finally, whether raw data is generated through testing or modeling, it will not always be possible to predict all scalability problems. Sometimes, a system gives no sign prior to a scalability failure, as when this failure is caused by a bug. As stated by Weinstock and Goodenough, an analysis can only give greater confidence in the scalability of the system, not an absolute guarantee.

Contributions and Benefits

The contributions associated with this chapter are as follows:

Contribution: *a framework for characterizing and analyzing software system scalability that is independent of the application domain and system design; and*

Contribution: *an analysis technique for evaluating and comparing scalability characteristics of software systems.*

The scalability framework (Figure 4.1) is based on concepts that are common to software systems in general, such as quality goals and system metrics, and makes no assumptions on the application domain or system design. Furthermore, in Section 4.3.2, we described two common objectives when evaluating or comparing the scalability of software systems: (1) checking whether the system can meet its quality goals when characteristics of its application domain and system design scale and (2) exploring the system's limits and scaling trends. In the scalability framework, quality goals and their relative importance are described in terms of preference and utility functions over dependent variables, while the characteristics of its application domain and system design are independent variables. How the analysis will answer questions related to the above objectives depends on the semantics chosen for variables and functions. For example, preferences and utility functions can be modeled in such a way that a zero value indicates no compliance with quality goals, so that objective (1) can be achieved by checking whether the preference or utility curve reaches zero at any point of the scaling range of independent variables. Similarly, in order to achieve objective (2), independent variables can be scaled until a zero utility is reached, uncovering the limits of the system. The scaling trends can be characterized both in terms of the values assumed by dependent variables or, more meaningfully, in terms of the system's preferences and utility.

An additional benefit of the concepts introduced in this chapter is:

Benefit: *a precise and uniform vocabulary that stakeholders can use to articulate scalability concerns.*

Numerous are the cases in the computing literature in which scalability is mentioned in passing or simply claimed outright but never fully defined or justified. The lack of a consensus on both the meaning of the term and on tools and techniques to characterize and analyze the scalability of software systems makes it difficult to describe the scalability of a system clearly, to evaluate claims of scalability and compare claims from different sources. Doing so requires an understanding of the system's scaling characteristics and their impact on quality goals. The scalability framework translates scalability questions and goals into clearly defined variables and functions, offering the stakeholder a precise and uniform vocabulary to articulate scalability claims. Furthermore, trends described by dependent variables, preferences and utility values give sufficient information for others to confirm or dispute these claims.

4.6 Summary

The purpose of a scalability analysis is to explain the effects that variations within characteristics of the application domain (e.g., the number of concurrent requests) and variations within characteristics of the software design (e.g., the number of machines in a cluster) have on the levels of satisfaction of quality goals (e.g., goals on response time or resource usage). The analysis uses preference functions for modeling quality goals and utility functions for aggregating multiple preferences into a single value.

However, the wrong choice of variables and functions can compromise the analysis results. The next chapter presents a technique for elaborating the system's scalability goals, from which the framework variables and functions are ultimately derived.

Chapter 5

Elaborating Scalability Requirements

Despite the recognized importance of considering scalability since the early stages of development, there is currently little support for reasoning about scalability at the requirements level. This chapter presents a goal-oriented approach for modeling and reasoning about scalability requirements.

5.1 On Scalability and Requirements Engineering

A system's ability to scale is strongly dependent on decisions taken at the requirements and architecture levels (Bahsoon and Emmerich, 2008). Failure to consider scalability in these stages may result in systems that are impossible or too costly to change when scalability problems become apparent during testing or system usage. Scalability is one of the most critical requirements for software systems (Gabriel et al., 2006) and should be taken into consideration from the very early stages of development. Yet, despite the importance of scalability to software engineering, there is currently a lack of systematic methods for modeling and reasoning about scalability requirements.

Our discussions with developers in industry revealed that they generally paid no systematic attention to scalability during requirements engineering (Industry Interviews, 2007). Many companies concentrated on functional requirements, never considering load, growth characteristics, or technical boundaries when designing their systems. In fact, some companies, struggling to gain market share and generate revenue, believed concentrating on functionality was the only possible way to succeed. Although time-to-market concerns are arguably justifiable for some systems, ignoring non-functional requirements may lead to scalability problems that cannot be fixed quickly, and the repeated use of workarounds can make the system unmanageable.

When scalability was not completely ignored, requirements were often described in terms of the application domain boundaries only—a limited approach as discussed in Chapter 1. Sometimes, scalability requirements considering different points on the range application domain characteristics can be drawn naturally from Service Level Agreements (SLA). For example, in an e-commerce system, a slight drop in performance might be acceptable during high volume periods, such as Christmas. We however have also seen scalability requirements that appeared to be derived from folklore or the willingness to comply with generally accepted good practices in software development. Such requirements can impose unjustifiable demands on the system design. This is the case of the vague belief in the need for linear scalability, such as when throughput is required to increase linearly with machine capacity (Brataas and Hughes, 2004).

Conflicts of interest between management and technical groups were common. Management often pushed for a quick solution, even when scalability problems had been predicted. As a result, people tended not to take responsibility for scalability. With the quality neglected or stated in vague terms only, developers had no clearly defined, justifiable targets to ensure the system meets its scalability goals. Consequently, workload tests were largely overlooked and scalability problems became apparent only during production.

5.1.1 Challenges in Elaborating Scalability Requirements

In Chapter 3, scalability was defined as *the ability of a system to satisfy its quality goals to levels that are acceptable to its stakeholders when characteristics of the application domain and the system design vary over expected ranges*. The precise definition of a system's scalability requirements is, therefore, relative to other primary quality goals for the system. For example, the scalability of a Web search engine may be characterized by its ability to return search results in a time that remains almost instantaneous—*a perfor-*

mance goal—when the number of simultaneous queries and the number of pages indexed increases; or the scalability of an air traffic control system may be characterized by its ability to keep safe separation distances between airplanes—*a safety goal*—when the number of airplanes simultaneously present in the airspace region increases.[1]

Scalability requirements should be identified early, and defined in a precise, testable way. The task, however, is not trivial. Some of the main challenges in elaborating scalability requirements are:

1. Often scalability problems come from unforeseen circumstances and system exceptions as described in Chapter 4. Elaborating scalability requirements demands a systematic exploration of unusual scenarios and of the load imposed by exceptions triggered during the system execution.

2. Scalability is a multivariate problem, which is hard to solve unassisted. If too many application domain and design characteristics are taken into consideration, there is the risk of the problem becoming too hard and prohibitively time-consuming to solve or understand. There is also the risk of the scalability requirements simply reflecting the intended system design—when the opposite should be true. Furthermore, each system quality is normally related to a number of characteristics in the application domain and system design. Even if the scaling of a particular characteristic does not obstruct the satisfaction of a quality goal, the combined scaling of multiple characteristics may do so. Conversely, multiple qualities may be affected by the scaling of a single characteristic. Elaborating scalability requirements demands uncovering the relationship between scaling characteristics and system qualities; and exploring the combined effect of these characteristics on each system quality.

3. Characterizing the possible range of values that application domain characteristics may assume can be challenging. In fact, in the early stages of the software development, the application domain and its growth characteristics are often not sufficiently known. Exceptions are driven by the industry in which the company finds itself; in industries where the revenue depends on the input volume handled, companies are usually more careful about characterizing the application domain and establishing SLAs.

4. Scalability makes even blurrier the boundary between requirements and design. Scalability is frequently achieved by means of a design-based scaling strategy, where the system capacity is varied to deal with the scaling of the application domain (Weinstock and Goodenough, 2006). Therefore, it is not always possible, during requirements engineering, to verify the satisfiability of scalability goals. This inability may prevent the assessment of scalability risks during requirements engineering, requiring these goals to be clearly "marked" for later investigation.

[1]These are just simplified examples. The scalability of a system will be characterized with respect to multiple goals and scaling characteristics.

5.2 An Experience with KAOS

After a review of the available requirements engineering techniques, we settled on KAOS as a good candidate for elaborating scalability requirements and investigated its suitability by applying the technique to the IEF. This experience was reported on (Duboc et al., 2008) and led us to recognize the benefits of KAOS that are particularly useful for scalability:

Makes assumptions explicit: The scalability of a system is highly dependent not only on the assumptions made about the current system environment, but also on the estimation of this environment in the future. Systems may reach scalability barriers because such assumptions were based on incorrect premises. Making such assumptions explicit is crucial for verifying premises and justifying scalability requirements.

Provides rationale for requirements: Without an explicit statement of their rationale, scalability requirements may impose unjustifiable demands on the system design, such as a unfounded goal for linear scalability. Goal models allow one to analyze the completeness and relevance of scalability requirements with respect to higher-level goals.

Provides traceability: Traceability is particularly useful in the context of scalability. If, in the future, assumptions on application domain ranges are found to be incorrect or no longer valid, they can be traced easily to the system requirements.

Provides assignment of responsibilities: By assigning goals to agents, KAOS can formally establish the responsibility for scalability. For example, assigning an assumption on the expected workload to the agents customer and service provider clearly establishes that both parties should agree on a supported workload that should be respected during production.

Provides measurable quality variables and objective functions: One can draw an almost direct parallel between KAOS's quality variables and objective functions and the scalability framework's variables and functions (Figure 4.1). Therefore, goal models can be used quite naturally as input to the scalability analysis, as is described in Chapter 6.

Despite these benefits, we found a number of difficulties in applying KAOS for elaborating scalability requirements. Most importantly, there is currently *a lack of sound techniques for elaborating scalability goals* (Duboc et al., 2008). In particular:

Handling ranges in the application domain: Although KAOS provides a means of identifying and specifying measurable quality goals as objective functions, it overlooks the existence of ranges and their probability distribution of values. Techniques are needed for the systematic elicitation and specification of such ranges in a goal model.

Taxonomy of application domain assumptions: While goals have a clear taxonomy (functional, performance, reliability, security, etc.), KAOS lacks a taxonomy for assumptions. Should an assumption on the range of a characteristic of the application domain be treated in the same way as an assumption on the correctness of the input format? Which elements should each category of assumptions contain? Such taxonomy and associated heuristics would help future modeling efforts.

Handling disagreements on range values: In the IEF, some of the "conflicts" between assumptions were not actually conflicts or divergences in the KAOS sense (van Lamsweerde, 2001), but rather disagreements about the range of values for application domain characteristics. KAOS lacks an appropriate way to model these disagreements in a KAOS goal tree. Modeling disagreements is important not only as a negotiation tool, but also to enable an analysis of the impact of different assumptions on goals.

Handling time-varying assumptions: It is unclear how to deal with assumptions that vary over time in KAOS. For example, the increase of electronic transactions and advances on hardware technologies mean that the assumptions on the IEFs application domain and required infrastructure will vary over time. KAOS lacks ways to express this variation in goal models.

Modeling hardware characteristics in goal trees: When the hardware capacity is varied to support the scaling of application domain characteristics, it is unclear whether a distinction between hardware and software should be made in a goal model. For the IEF, we explicitly chose to model assumptions on hardware infrastructure because such strategy imposes a requirement on the predictability of system performance in various infrastructures, and because the responsibility for software and hardware were spread over two organizations (banks were required to provide the infrastructure to run the IEF).

Based on the conclusions drawn from the industry interviews, discussions with other computer scientists and this initial experience with KAOS, we developed a technique for elaborating scalability requirements. The technique inherits KAOS advantages, while tackling some of the challenges and drawbacks discussed above.

5.3 Specifying Scalability

In our definition of scalability, the elements required to define what it means for a system to be scalable are the quality goals of the system; the characteristics of the application domain and system design that are expected to vary and their ranges; and the acceptable levels of quality goal satisfaction under these variations. In the KAOS framework, these elements can be represented as follows:

1. Quality goals correspond to goals whose specification includes domain-dependent objective functions.

2. Expected variations of application domain characteristics are represented in domain assumptions that we call *scaling assumptions*.[2]

[2]At the requirements level, scaling assumptions are only concerned with variations of characteristics in the application domain (Zave and Jackson, 1997).

3. The acceptable levels of quality goal satisfaction under these variations can be specified as target values for the quality goal objective functions. We refer to quality goals that are constrained by scaling assumptions as *scalability goals*.

The following sections define the concepts of scaling assumption and scalability goal in more detail, and illustrate their uses and roles in the elaboration of software requirements.

5.3.1 Specifying Scaling Assumptions

We define a *scaling assumption* as a domain assumption specifying how certain characteristics in the application domain (i.e., domain quantities) are expected to vary over time and/or across different categories of system instances. The specification of scaling assumptions should make reference to:

1. one or more *domain quantities* whose expected variations are defined in the assumption;
2. the *periods of time* and *categories* of system instances over which the assumption is defined; and
3. the *ranges of values* each quantity is expected to take for each system category over each period of time.

For example, in the IEF, bank transactions can be submitted for processing in daily batches. The following scaling assumption specifies how the number of transactions in daily batches submitted to the IEF is expected to vary over time and for different categories of banks; Banks can be classified in categories as small, medium, large and merge (acquisition of one bank by another), as in Assumption 5.1.

Assumption 5.1

Assumption Expected Batch Size Evolution
Category Scalability
Definition Between 2009 and 2015, daily batches are expected to contain up to the following numbers of transactions for the different bank categories:

Bank	**2009**	**until 2012**	**until 2015**
small	10,000	15,000	20,000
medium	1 million	1.2 million	1.8 million
large	50 million	55 million	60 million
merger	80 million	85 million	95 million

Our definition of scaling assumption covers the case where variations of domain quantities refer to a single period of time and a unique set of system instances. For example, the scaling of the IEF's batch size can be described as in Assumption 5.2.

Assumption 5.2

Assumption Expected Batch Size Variation
Category Scalability
Definition Over the next three years, daily batches of transactions for all our customers are expected to vary between 10,000 and 95 million transactions.

Scaling assumptions can be specified with varying precision. For some projects, it may be useful to distinguish between the average and peak value of some domain quantity. The specification of a scaling assumption might even refer to the probability distributions of the domain quantities under consideration (e.g., the distribution of the number of transactions in a daily batch).

Eliciting information about domain quantities might be challenging. Such information may come from domain knowledge, from workload characterization of other systems in the same domain, from similar domains, among others. In the IEF, for instance, the number of transactions per account per day and the percentage of transactions that are fraudulent are known in the financial domain. In an ambulance dispatching system, data collected over the years allow to estimate the number of emergency calls the system is expected to receive. However, particularly during the early stages of the development lifecycle, there may be many uncertainties about the expected variations of some domain characteristics. When this is the case, it is vain to try to elicit precise numbers and detailed probability distributions from stakeholders. Fortunately, such level of detail is not always needed; identifying orders of magnitude for all domain quantities might be enough to inform the system and software design decisions adequately. Nevertheless, just because there are uncertainties, it does not mean that scalability requirements should be specified in vague terms. A formalization technique is important to analyze this uncertainty and understand it better.

Note that scaling assumptions are *not* requirements; they describe properties that are assumed to be true in the application domain rather than properties that must be enforced by the software-to-be. Scalability requirements are specified by defining the required levels of satisfaction for other quality requirements under the variations given in the scaling assumptions, as shown below.

The Role of Scaling Assumptions

Semantically, scaling assumptions express constraints on the ranges of values that domain quantities are expected to take over the specified time periods and categories of instances. Logically, the *absence* of a scaling assumption for a domain quantity means that there is *no assumed constraint* on its possible values; that is, potentially its value at any point in time could be infinite.

Scaling assumptions play a similar role in software development as other kinds of domain assumptions, that is, they support the refinement of goals towards requirements and expectations that can be satisfied by a single agent (Zave and Jackson, 1997; van Lamsweerde, 2008). More precisely, scaling assumptions support the refinement of quality goals expected to handle an unbounded number of domain quantities. To illustrate this, consider the IEF goal Achieve[Batch Processed Quickly] (Goal 5.1) below [3] The natural language and formal definitions state that the goal is satisfied in an absolute sense if every batch of transactions submitted is processed in less than 8 hours. The quality variable processingTime is a random variable whose sample space is the set of batches submitted to the IEF. Its value denotes the time to process a batch; that is, the time to verify all transactions in the batch and generate alerts for those that seem fraudulent. The objective function states that the system should be designed so as to maximize the

[3]This and other goals in this thesis are stated examples to demonstrate use of the technique. Different parameters applied to the actual system.

probability that a batch is processed in less than 8 hours. The “Must” column states that this percentage should be at least 90%. The goal is assigned to the Alert Generator agent, a subsystem of the IEF.

Goal 5.1

Goal Achieve [Batch Processed Quickly]

Category Performance

Definition Daily batches of transactions provided by the bank should be processed in less than 8 hours.

Quality Variable: processingTime: Time

Definition: The time required to process the batch

Sample Space: Set of batches submitted

Objective Functions: At least 90% of the batches should be processed within 8 hours, and all batches should be processed within 9.6 hours.

Name	Definition	Mode	Must
% of batches processed in 8 hours	Pr(processingTime $\leq$ 8h)	Max	90%
% of batches processed in 9.6 hours	Pr(processingTime $\leq$ 9.6h)	Max	100%

Responsibility Alert Generator

The goal is *realizable* by the agent as it is defined entirely in terms of quantities that are monitored and controlled by that agent (Letier and van Lamsweerde, 2002; Zave and Jackson, 1997; Courtois and Parnas, 1993). Realizability is a theoretical notion of what it means for a goal to be achievable by an agent based on the agent’s monitoring and control capabilities only. It implicitly assumes that the agent has infinite capacity, where *agent capacity* denotes a domain-specific measure intended to characterize the amount of resources available to the agent to satisfy the goal. In practice, however, all agents have finite capacities. It is impossible for the Alert Generator to guarantee the satisfaction of the goal irrespectively of the size of the daily batches to be processed.

In order to obtain a goal whose satisfaction can be guaranteed to be realized by the Alert Generator alone, the goal Achieve[Batch Processed Quickly] can be refined into a scaling assumption on the size of the transactional batches, such as in Assumptions 5.1 and 5.2, plus a goal requiring daily batches to be processed within 8 hours *only for batches satisfying the scaling assumption*. Using Assumption 5.1, we obtain the Goal 5.2.

Goal 5.2

Goal Achieve [Batch Processed Quickly Under Expected Batch Size Evolution]

Category Performance and Scalability

Definition At the end of each day, the batch of transactions submitted by the bank should be processed in less than 8 hours, provided that the batch size does not exceed the bounds stated in the scaling assumption ‘Expected Batch Size Evolution’.

This goal now can be satisfied by an Alert Generator with finite capacities (i.e., finite processing speed, memory and storage capacities).

We ignore for the moment the specification of the partial levels of satisfaction using objective functions and how to decide which scaling assumption to apply to a given goal. These will be considered in the next sections.

5.3.2 Specifying Scalability Goals

We define a *scalability goal* as a goal whose specification includes a description of the required levels of the goal satisfaction under variations of some application domain quantities, as specified in one or more scaling assumptions. Goal 5.2 is an example of a scalability goal.

A frequent type of scalability goal is one in which the same level of goal satisfaction must be maintained under the full range of variations specified in the associated scaling assumptions. We call these *scalability goals with fixed objectives*. These are goals for which the definition, objective functions and target values are the same across the whole range of values considered in the scaling assumptions. For example, with fixed objectives, the specification of the scalability goal Achieve[Batch Processed Quickly Under Expected Batch Size Evolution] will be as in Goal 5.3. The objective function is defined as the conditional probability that a batch is processed in less than 8 hours when it satisfies the scaling assumption Expected Batch Size Evolution.

Goal 5.3

Goal Achieve [Batch Processed Quickly Under Expected Batch Size Evolution]
Category Performance and Scalability
Definition At the end of each day, the batch of transactions submitted by the bank should be processed in less than 8 hours, provided that the batch size does not exceed the bounds stated in the scaling assumption 'Expected Batch Size Evolution'.
Quality Variable: processingTime: Time
Definition: The time required to process the batch
Sample Space: Set of batches submitted
Objective Functions: At least 90% of the batches should be processed within 8 hours, and all batches should be processed within 9.6 hours.

Name	Definition	Mode	Must
% of batches processed in 8 hours	Pr(processingTime $\leq$ 8h | ExpectedBatchSizeEvolution)	Max	90%
% of batches processed in 9.6 hours	Pr(processingTime $\leq$9.6h | ExpectedBatchSizeEvolution)	Max	100%

Responsibility Alert Generator

By contrast, *scalability goals with varying objectives* are scalability goals whose required level of goal satisfaction is not the same under the whole range of variations specified in the scaling assumptions. Their goal definition, objective functions definitions, or target values are different across the range of values considered in the scaling assumptions. For example, a specification of the goal Achieve[Batch Processed Quickly Under Expected Batch Size Variation] with varying objectives is as in Goal 5.4.

Goal 5.4

Goal Achieve[Batch Processed Quickly Under Expected Batch Size Evolution]

Category Performance and Scalability

Definition At the end of each day, if the size of the batch submitted by the bank conforms to constraints stated in the scaling assumption "Expected Batch Size Evolution", the whole batch must be processed in less than X hours where the value of X is specified in the following table:

Bank	**2009**	**2010 - 2012**	**2013 - 2015**
small	4h	4h30m	5h
medium	5h	5h30m	6h
large	8h	8h30	9h
merger	10h	10h30	11h

Formal Spec (∀ b:Batch)

$\Box$ (b.Submitted $\rightarrow \Diamond_{\leq X hours}$ b.Processed) [a]

Quality Variable processingTime : Time

Objective Function: At least 90% of the batches should be processed within X hours and all batches should be processed in 1.2 X hours (tolerance of 20% in the processing time).

Name	**Definition**	**Mode**	**Must**
% of batches processed in X hours	*Pr*(processingTime $\leq$ X h | ExpectedBatchSizeEvolution)	Max	90%
% of batches processed in 1.2 X hours	*Pr*(processingTime $\leq$ 1.2 X h | ExpectedBatchSizeEvolution)	Max	100%

Responsibility: Alert Generator

[a]See Section 2.1.2 for explanation of syntax.

Scalability goals with varying objectives can also be used to specify how system qualities are allowed to degrade under exceptional conditions. In particular, they can specify "graceful" degradation of goal satisfaction when the system operates outside of its normal scaling ranges, a desired property of software systems.

Scalability goals can be specified at different levels of abstraction in the goal refinement graph. Following the KAOS definitions, a *scalability requirement* is a scalability goal assigned to an agent in the software-to-be, while a *scalability expectation* is a scalability goal assigned to an agent in the environment.

5.4 A Systematic Process

This section presents a *systematic process* to guide the modeling and analysis of scalability concerns during goal-oriented requirements engineering.

5.4.1 Process Overview

In Chapter 3, we explained that what it means for a system to be scalable is relative to other quality goals for the system. Identifying the primary quality goals of a system is therefore a prerequisite to the precise specification of its scalability goals. The process described in this section assumes that functional and quality goals of the system have been elaborated following a systematic KAOS process (van Lamsweerde, 2008). During such an elaboration process, the initial specification of goals, and the assignment of goals to agents, are typically idealized. The model is generally elaborated without explicit consideration for the scaling characteristics of the application domain and the limited capacities of agents.

Starting from idealized goals is beneficial as it avoids premature compromises based on implicit and possibly incorrect stakeholder perceptions of what might be achievable. However, later on in the requirements elaboration process, it is necessary to take into account what can be achieved realistically by agents and, if needed, modify the specifications of goals, requirements, and expectations accordingly.

Our process for elaborating scalability requirements is shown in Figure 5.1. This process is based on the goal-obstacle analysis loop of the KAOS method, which is a goal-anchored form of risk analysis (van Lamsweerde and Letier, 2000). Starting from an initial goal model, the goal-obstacle analysis loop consists of:

1. systematically *identifying obstacles* that may obstruct the satisfaction of the goals, requirements, and expectations elaborated so far;
2. *assessing* the likelihood and criticality of those obstacles; and
3. *resolving* them by modifying existing goals, requirements, and expectations, or generating new ones so as to prevent, reduce or mitigate the identified obstacles.

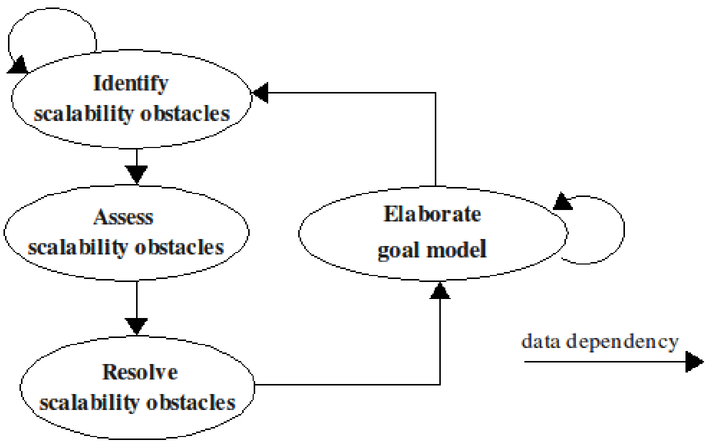

Identify scalability obstacles
Assess scalability obstacles
Resolve scalability obstacles
Elaborate goal model
data dependency

Figure 5.1: High-level process for elaborating scalability requirements.

Goal-obstacle analysis is performed iteratively until all remaining risks are considered acceptable. The application of these steps to handle scalability concerns will lead to the identification and assessment of scalability obstacles and to their resolution. Some of the model elaboration tactics for resolving scalability obstacles lead to the identification of scaling assumptions and scalability goals introduced in Section 5.3. We describe each of these three steps in the following subsections.

5.4.2 Identifying Scalability Obstacles

As discussed in Chapter 3, *load* and *capacity* are fundamental drivers of scalability.[4]. We therefore define a *scalability obstacle* as a condition that prevents some goals from being satisfied because the load imposed by the goals on agents involved in their satisfaction exceeds the capacity of those agents. Therefore, a scalability obstacle will be of the form $\lozenge(Load_G > Capacity_{ag})$. The concepts of goal load and agent capacity are defined as follows:

Goal load: a domain-specific measure intended to characterize the amount of work needed to satisfy the goal.
Agent capacity: a domain-specific measure intended to characterize the amount of resources available to the agent to satisfy its goal.

In order to identify concrete scalability obstacles, goal loads and agent capacities must be instantiated to measures that are specific to the goals and agents being considered. For example, a scalability obstacle to the goal Achieve[Batch Processed Quickly] (Goal 5.1) is the condition Batch Size Exceeds Alert Generator Processing Speed. The goal load is the number of transactions in the batches to be processed, and the agent capacity is the Alert Generator's processing speed measured as the number of transactions it can process per hour. As another example, consider the downstream process of investigating the alerts generated by the IEF. Once an alert is generated, fraud investigators are expected to verify the alert within the next day, confirming or dismissing the suspicion of fraud. This goal, named Achieve[Alerts Investigated Quickly], would have as a scalability obstacle the condition Number of Alerts Exceeds Fraud Investigators Speed. In this case, the goal load is the number of alerts to be processed each day and the agent capacity is the number of alerts the team of fraud investigators is able to investigate per day.

A scalability obstacle can be an obstacle to goals at any level in the goal refinement structure. However, during scalability obstacle analysis, we find it preferable to generate scalability obstacles from lower level goals assigned to single agents because it allows one to generate more specific obstacles and investigate more specific resolution strategies than by considering scalability obstacles to higher level goals. That is, one should consider scalability obstacles to all requirements and expectations in the model. Note that domain properties and domain hypothesis cannot be obstructed by scalability obstacles, as they are not under the responsibility of an agent. However, if incorrect, they may cause scalability obstacles to other goals in the model. For example, if the assumption Expected Batch Size Evolution (Assumption 5.1) is violated in the running system, it may create a scalability obstacle to the goal Achieve[Batch Processed Quickly] (Goal 5.1).

Patterns of scalability obstructions

The generation of obstacles from goals in (van Lamsweerde and Letier, 2000) is guided by formal obstruction patterns. In order to help the generation of scalability obstacles, we have created scalability

[4]There might be others, as is discussed in in Section 8.4. Load is determined by the scaling of application domain characteristics associated with goals. The agent's ability to support the load imposed by the goals under its responsibility will depend on its capacity.

obstruction patterns in Table 5.1. [5]

Goal	**Domain Property**	**Obstacle**
G: $\Box$P	$\Box$(P $\rightarrow$ $Load_G \leq Capacity_{ag})$	$\Diamond(Load_G > Capacity_{ag})$
G: $\Box$(A $\rightarrow$ P)	$\Box$(P $\rightarrow$ $Load_G \leq Capacity_{ag})$	$\Diamond$(A $\wedge$ $Load_G > Capacity_{ag})$

Table 5.1: Scalability obstruction patterns.

The first pattern considers goals of the form $\Box$P, which do not refer to a scaling assumption.[6] It uses a domain property stating that a necessary condition for the goal to be satisfied is that the goal load is less than or equal to the agent capacity. The identified scalability obstacle has the form Goal Load Exceeds Agent Capacity. Take, for example, the IEF goal Achieve[Batch Processed Quickly] (Goal 5.1) and its assignment to the Alert Generator. This goal is impossible to satisfy without some bound on the size of the transactional batches submitted to the system. The application of the first pattern on this goal generates the scalability obstacle Batch Size Exceeds Alert Generator Processing Speed whose formal definition is given by: $\Diamond$ ($\exists$ b:Batch, ag:AlertGenerator) Size(b) $>$ Speed(ag).

The second pattern considers goals of the form $\Box$(A $\rightarrow$ P), where A is the condition defined in a scaling assumption on the goal load.[7] It uses the same domain property and produces a scalability obstacle of the form Goal Load is within Scaling Assumption and Exceeds Agent Capacity. Take, for example the IEF goal Achieve[Batch Processed Quickly under Expected Batch Size Evolution] (Goal 5.2). A scalability obstacle occurs if the system does not have sufficient capacity to process a transactional batch, even if its size is within the expected bounds. The application of the second pattern on this goal generates the obstacle Batch Size is within Expected Evolution and Exceeds Alert Generator Processing Speed whose formal definition is given by: $\Diamond$ ($\exists$ b:Batch, ag:AlertGenerator) WithinExpectedSize(b) $\wedge$ (Size(b) $>$ Speed(ag)).

However, when identifying scalability obstacles, it is not sufficient to consider obstacles to each goal in isolation. There might be situations where an agent capacity might be sufficient to satisfy each goal in isolation but insufficient when having to satisfy all goals together. The additional obstruction pattern in Table 5.2 can be used to generate scalability obstacles to multiple goals.

Goals	**Domain Property**	**Obstacle**
G1,...,Gn	G1 $\wedge \ldots \wedge$ Gn $\rightarrow$ $\Box(Load_{G1} + \ldots + Load_{Gn} \leq Capacity_{ag})$	$\Diamond(Load_{G1} + \ldots + Load_{Gn})$ $> Capacity_{ag}$

Table 5.2: Pattern of scalability obstruction to multiple goals.

Consider a set of goals $G1, ..., Gn$ assigned to a same agent ag, and a domain property $G1 \wedge ... \wedge Gn \implies \Box(Load_{G1} + ... + Load_{Gn} \leq Capacity_{ag})$ stating that the agent's capacity must always be

[5]See Section 2.1.2 for explanation of syntax.

[6]A goal of the form textsf$\Box$P states that the condition textsfP must be true for all future states.

[7]A goal of the form $\Box$(A $\rightarrow$ P) states that always, if the condition A is true then the condition P must also be true.

bigger than the sum of the goals' loads in order for the goals to be satisfied. A scalability obstacle to this set of goals would be the condition $\Diamond(Load_{G1} + ... + Load_{Gn} > Capacity_{ag})$, where the sum of the goal's load eventually exceeds the agent capacity. When $n = 1$, this pattern is equivalent to the first obstruction pattern to a single goal in Table 5.1.

To illustrate this pattern, consider the following two IEF goals: Achieve[Incoming Transactions Stored] and Achieve[Alerts Stored]. These goals state, respectively, that all transactions submitted to the system and all alerts generated by the system should be stored in the system. Both goals are assigned to the agent Data Storage, as illustrated in Figure 5.2. A necessary condition for both goals to be satisfied is that the sum of the transactions and alerts data (goal load) does not exceed the Data Storage's disk space (agent's capacity). An application of the pattern generates the scalability obstacle Amount of Data Exceeds Data Storage Space. Note that the strict application of the pattern for a single goal described in the beginning of this section would have generated two obstacles: Number of Incoming Transactions Exceeds Data Storage Space and Number of Alerts Exceeds Data Storage Space. The logical conjunction of these two scalability obstacles is not equivalent to the joined scalability obstacle Amount of Data Exceeds Data Storage Space.

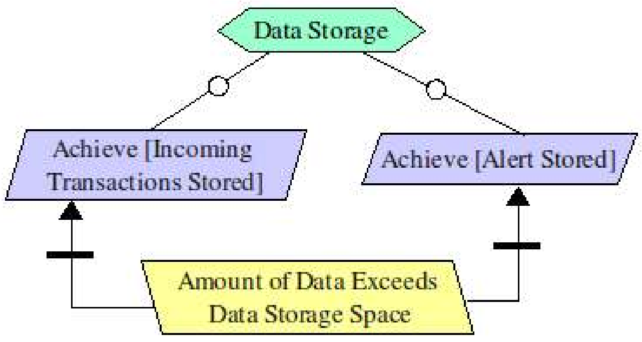

Data Storage
Achieve [Incoming Transactions Stored]
Achieve [Alert Stored]
Amount of Data Exceeds Data Storage Space

Figure 5.2: Scalability obstacle to multiple goals.

In the KAOS framework, the situation described in this pattern can also be viewed as a *divergence* between the set of goals $G1, ..., Gn$ where the scalability obstacle is the *boundary condition* for this divergence (van Lamsweerde et al., 1998). For the purpose of elaborating scalability requirements, we found it more convenient to treat such situations as obstructions rather than divergences because our main focus is on identifying and resolving scalability problems rather than handling goal conflicts. Identifying and resolving these scalability obstacles will also resolve the goal divergences as a by-product.

Using obstruction patterns

The scalability obstruction pattern to multiple goals suggests the following process to identify scalability obstacles: For each agent,

1. identify the set of all goals assigned to the agent;
2. for each goal, identify what defines the goal load and what agent resources are involved in its satisfaction;
3. for each agent resource M identified in step 2, if it is a domain property that the satisfaction of

all goals involving the resource M requires that the sums of the goals loads is always smaller or equal to the amount of resource M of the agent, then generate a scalability obstacle of the form $\Diamond(Load_{G1} + ... + Load_{Gn} > M)$ obstructing the set of goals $G1, ..., Gn$.

This process is an over-simplification, as it assumes that the agent alone is responsible for the satisfaction of all goals under consideration and that the total load imposed on an agent can be calculated as the addition of individual goal loads. The pattern of scalability obstruction for multiple goals could be generalized to a situation where the domain property involves any monotonically increasing function $F(Load_{G1}, ..., Load_{Gn})$ instead of a simple addition. However, further research is needed to to create a process that considers interactions between agents and/or goals and more complex calculation of the total load imposed on agents, as will be discussed in Chapter 8. Furthermore, it may not be interesting to consider specific resources (such as CPU, memory, network bandwidth or disk) at early stages of requirements specification, as satisfying a goal probably will use portions of a number of different resources. However, if an agent is responsible for goals that will be critical to specific resources, then it might be interesting to analyze these resources separately. The technique allows one to model resources at different levels of abstraction, and the appropriate level depends on the particular goal.

5.4.3 Assessing Scalability Obstacles

Once potential scalability obstacles have been identified, their criticality and likelihood should be assessed. As for other kinds of obstacles, criticality depends on its impact on higher-level goals and the criticality of those goals.

A lightweight technique to support this process consists of using a standard *qualitative risk analysis matrix* in which the likelihood of a scalability obstacle is estimated on a qualitative scale from *Very Unlikely* to *Almost Certain* and its criticality is estimated on a scale from *Insignificant* to *Catastrophic* (van Lamsweerde, 2008). The goal refinement graph helps estimating obstacle criticality by allowing one to follow goal-refinement links upward to trace all high-level goals affected by an obstacle.

When needed, a more detailed, quantitative analysis of the impact of obstacles on higher-level goals could be performed using quality variables refinement equations of a quantitative goal model (Letier and van Lamsweerde, 2004). Much further work is needed in order to develop techniques for a quantitative scalability analysis at the requirements and goal levels, as it will be discussed in Chapter 8. It is important to note, however, that a detailed, quantitative analysis of scalability during requirements elaboration is not the purpose of this work. In the early stages of scalability requirements elaboration, the main purpose of obstacle assessment is to separate scalability obstacles for which resolutions need to be sought from those whose risk is so low that they can safely be ignored. When possible, obstacles will be assessed and resolved at the goals level. When assessing the likelihood of some obstacle is not viable at requirements elaboration time, its obstructed goals should be clearly marked for later investigation. This can be easily accomplished in KAOS using the "issue" feature in goals, which is a process-level feature capturing questions raised during requirements elaboration but that can only be addressed at later stages of the development process.

5.4.4 Resolving Scalability Obstacles

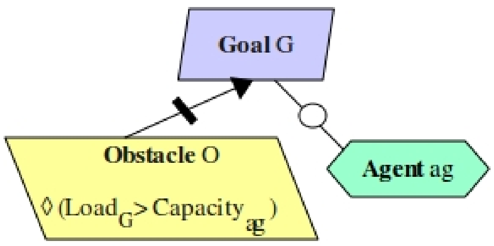
Scalability obstacles whose combined likelihood and criticality are considered serious enough must be resolved. The obstacle resolution process comprises two activities: the *generation* of alternative resolutions and the *selection* of resolutions among the generated alternatives. Only the generation step was investigated in this research; the selection among alternatives is left for future work.

This section presents a catalog of model transformation tactics for generating alternative ways to prevent, reduce or mitigate scalability obstacles. We have developed tactics for resolving scalability obstacles by systematically considering specializations along the eight general obstacles resolutions strategies in van Lamsweerde (2008): goal substitution, agent substitution, obstacle prevention, goal weakening, obstacle reduction, goal restoration, obstacle mitigation, and do-nothing. Tactics in our catalog formalize, at the goal level, strategies commonly used in industry to achieve scalability. They were collected from the industry interviews and from the computing literature. The benefit of a catalog of requirements-level scalability obstacle resolution tactics is to encode knowledge about how to deal with scalability risks at the requirements level in a form that allows system designers to explore a range of alternative system designs. Our catalog of scalability obstacle resolution tactics is summarized in Table 5.3. The first and second columns show the general obstacle resolution categories and descriptions, as defined in van Lamsweerde and Letier (2000). The third column shows the specialized tactics and sub-tactics for resolving scalability obstacles, respectively preceded by the black and white circles. The ones preceded by a dash are typical strategies to satisfy the goals introduced by the tactics and sub-tactics.

The remaining of this section presents these tactics in more detail. For each of the general obstacle resolution strategies, we briefly discuss its relevance for resolving scalability-related risks and, where appropriate, define specialized model transformation tactics that are specific to the resolution of scalability obstacles. Note that different tactics may be applied to resolve the same scalability obstacle, generating a set of alternative resolutions. The choice among these resolutions is made in the selection activity of the obstacle resolution phase.

All scalability obstacle resolution tactics have a scalability obstacle `O` as a precondition:

Precondition: A goal `G` is unsatisfiable by an agent `ag` because the load imposed by all goals assigned to `ag` exceeds the `ag`'s capacity.



Category	**Short Description**	**Scalability Obstacle Resolution Tactic**
Goal Substitution	Eliminates the need for the obstructed goal	(Direct application)
Agent Substitution	Assigns the responsibility for the obstructed goal to another agent	• Transfer goal to non-overloaded agent ○ Automate responsibility of overloaded human agent • Split goal load among multiple agents ○ Split goal load into subtasks ○ Split goal load by case
Obstacle Prevention	Avoids the obstacle entirely	• Set agent capacity according to load ○ Set agent capacity upfront to worst-case load ○ Adapt agent capacity at runtime according to load - Increase number of agent instances - Increase the capacity of the agent instance • Limit goal load according to agent capacity ○ Limit goal load according to fixed agent capacity ○ Limit goal load according to varying agent capacity - Limit goal load over time
Obstacle Reduction	Reduces its likelihood by introducing some ad-hoc countermeasure	(Direct application) - Influence goal load distribution
Goal Weakening	Weakens the obstructed goal specification	• Introduce scaling assumption • Strengthen scaling assumption • Weaken goal objective function ○ Relax real-time requirement ○ Relax required level of satisfaction
Goal Restoration	Tolerates the obstacle while requiring the target condition of the obstructed goal to be restored	(Direct application) - Postpone excess goal load
Obstacle Mitigation	Tolerates the obstacle and mitigates its consequences	(Direct application)

Table 5.3: Scalability obstacle resolution tactics.

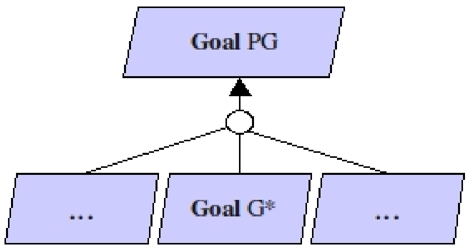
Each scalability obstacle resolution tactic transforms a model in which a scalability obstacle O obstructs a goal G under the responsibility of an agent ag into a new model that contains new or modified goals, domain assumptions, agents, or responsibility assignments. The application of several of these tactics results in the specification of new or modified scaling assumptions and scalability goals. The obstructed goal G may or may not already refer to scaling assumptions; that is, it may be of the form $\Box$P or $\Box$(A $\rightarrow$ P).

Goal Substitution

An effective way to resolve an obstacle to some goal is to eliminate the need for this goal by finding an alternative refinement to some higher-level goal in which the obstructed goal and its obstructing obstacle are no longer present. For scalability obstacles, the principle is to try to find an alternative way to satisfy higher-level goals in which the scalability risk no longer exists. Tactic 5.1 shows the refinement pattern for this resolution. As an example, consider the scalability obstacle Batch Size Exceeds Alert Generator's Processing Speed obstructing the goal Achieve[Batch Processed Quickly] (Goal 5.1). Historically, in the IEF, transactions were always processed in batches. A goal substitution is to consider a system that process all transactions continuously, 24 hours per day, rather than in overnight batches. [8]

Tactic 5.1: Goal substitution

Postcondition: The unsatisfiable goal `G` is eliminated by defining an alternative refinement for its parent goal `PG` involving subgoals different from `G` and such that `O` no longer exists.



Agent Substitution

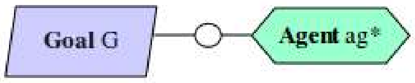
This tactic consists of eliminating the possibility for the obstacle to occur by assigning the responsibility for the obstructed goal to another agent. For resolving scalability obstacles, we define five new specializations of this tactic.

Transfer goal to non-overloaded agent: This tactic consists of transferring responsibility for one or more of the goals assigned to an overloaded agent to an alternative agent with bigger capacity or smaller load. Possible choices for the alternative agent can be explored systematically by considering agents already present in the model. This strategy may also lead to the introduction of a new agent role. A particular case is the commonly used strategy **automate responsibility of overloaded human agent**, in which a task normally accomplished by a human is automated by the system. The refinement pattern for this resolution is shown by Tactic 5.2.

[8]The current version of the system supports both processing options.

Tactic 5.2: Transfer goal to non-overloaded agent

Postcondition: Transfer the responsibility for the obstructed goal G from a single agent ag to another single agent ag* that alone has sufficient capacity to support the load imposed by the goals transfered from ag.



Split goal load among multiple agents: This tactic consists of refining a goal assigned to an overloaded agent into subgoals with smaller loads, such that each subgoal can be assigned to an agent with enough capacity to satisfy it. An application of this tactic may lead to the introduction of new agent roles taking responsibilities for the subgoals. The specialized tactics **split goal load into subtasks** and **split goal load by case** consists of generating the subgoals by refining the obstructed goal following a milestone-driven or a case-driven formal refinement pattern, respectively (Darimont and van Lamsweerde, 1996). The former identifies an intermediate condition that is a necessary milestone for reaching the target condition prescribed by the obstructed goal. It then creates a subgoal for reaching the milestone condition and another subgoal for reaching the target condition from the milestone condition, each assigned to an agent with sufficient capacity to support its goal load. The latter tactic identifies different cases for reaching the target condition. Each case is covered by a subgoal, whose disjunction implies the target condition of the obstructed goal. As in the former specialization, each subgoal is assigned to an agent with sufficient capacity to support its load. Tactics 5.3 and 5.4 shows the refinement patterns for these resolutions.

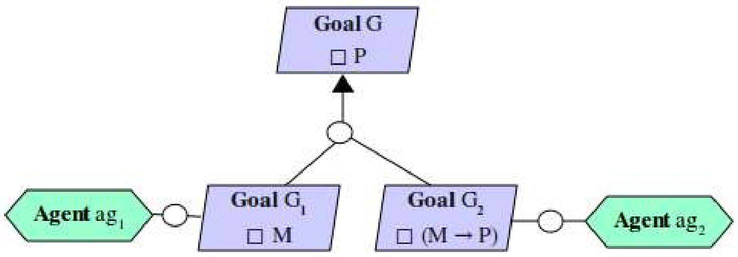
Tactic 5.3: Split goal load into subtasks

Postcondition: The unsatisfiable goal G is AND-refined into subgoals G_1 and G_2, such that:

(i) G_1 achieves an intermediate milestone condition for reaching the target condition prescribed by G,

(ii) G_2 achieves G's target condition starting from the intermediate milestone condition, and

(iii) G_1 and G_2 are respectively assigned to agents ag_1 and ag_2, who have sufficient capacity to support the load imposed by their assigned goals.



As an example in the IEF, consider the goal Achieve[Alerts Investigated Quickly] assigned to the agent Fraud Investigator, and the scalability obstacle Number of Generated Alerts Exceeds Fraud Investigator's Capacity. The tactic 'split goal load by case' suggests identifying a set of alternative cases of alert investigation that could be assigned to fraud investigators who specialize in such cases and are therefore more efficient to deal with them (e.g. based on the type of alert and category of the account holder). The tactic 'split goal load into subtasks' suggests identifying subgoals corresponding to subtasks that can be assigned to different agents in the system, such as assigning the responsibility of contacting card holders as part of the fraud investigation process to the agent Client Security Officer instead of Fraud Investigator.

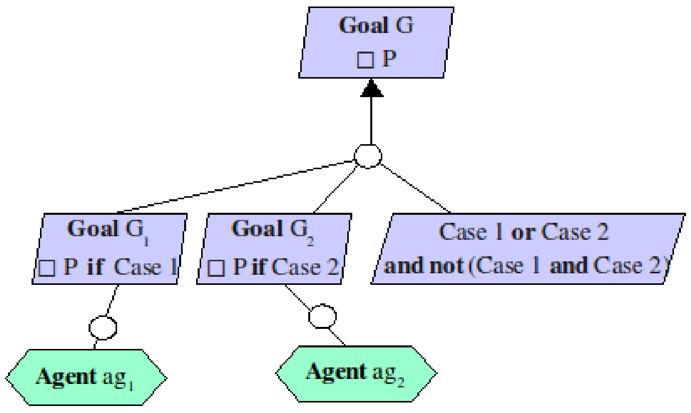
Tactic 5.4: Split goal load by case

Postcondition: The unsatisfiable goal $\texttt{G}$ is AND-refined into subgoals $\texttt{G}_1$ and $\texttt{G}_2$ that must be respectively satisfied in $\texttt{Case}_1$ and $\texttt{Case}_2$, such that:

(i) $\texttt{G}_1$ and $\texttt{G}_2$ each achieve the target condition prescribed by $\texttt{G}$,

(ii) $\texttt{Case}_1$ and $\texttt{Case}_2$ are disjoint and cover the entire state space, and

(iii) $\texttt{G}_1$ and $\texttt{G}_2$ are respectively assigned to agents $\texttt{ag}_1$ and $\texttt{ag}_2$, who have sufficient capacity to support the load imposed by their assigned goals.



Obstacle Prevention

This resolution tactic consists of generating a new goal requiring the obstacle to be avoided entirely. In the case of scalability obstacles, the generated goal has the following pattern:

Goal Pattern:

Goal Avoid [Goal Load Exceeds Agent Capacity]

Formal Spec $\square\neg(Load_G > Capacity_{ag})$

We have defined new goal refinement tactics and associated formal goal refinement patterns to guide the systematic exploration of alternative refinements for this goal:

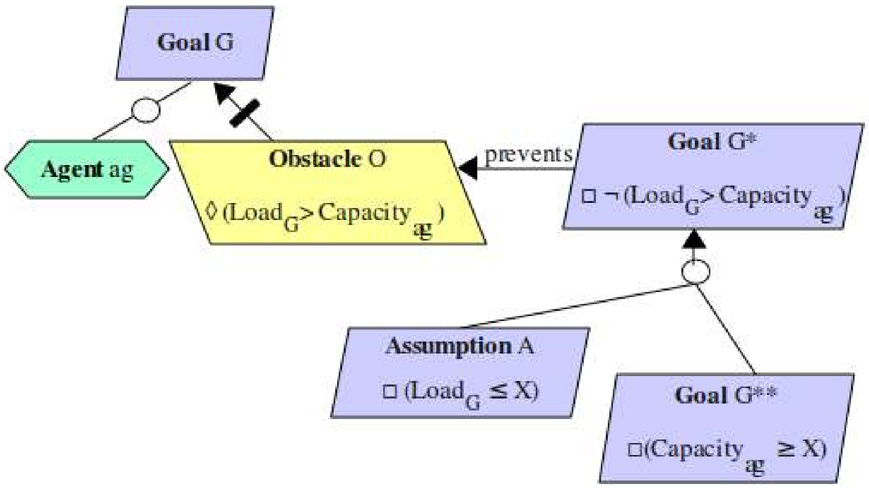
Set agent capacity according to load: This tactic consists of refining the above goal into a scaling assumption defining a bound X on the goal load, and a goal requiring the agent capacity to have a value equals or above X. Its refinement pattern is show in Tactic 5.5. This tactic has two specializations. On the specialization **set agent capacity upfront to worst-case load**, X defines a fixed worst-case bound on the goal load, and the goal requires the agent capacity to have a constant value equals or above this worst case load. The specialization **adapt agent capacity at runtime according to load** is a dynamic counterpart of the previous one. In this tactic, X defines a time-varying variable representing the predicted future value of the goal load at run-time, and the goal dynamically extends the agent capacity based on the predicted load. Two typical strategies to satisfy the subgoal of both specializations is to *increase the number of agent instances* so that their added capacity is greater then X and to *increase the capacity of the agent instance* so that it is greater than X. In the IEF, for example, that could be increasing the number of Fraud Investigators processing alerts or providing training to investigators so that alerts can be processed quicker. Tactic 5.4 shows the refinement for this resolution.

Tactic 5.5: Set agent capacity according to load

Postcondition: Define a new goal G* of the form $\square\neg(Load_G > Capacity_{ag})$ that is AND-refined into:

(i) a scaling assumption A of the form $\square(Load_G \leq X)$ defining a bound on G's load, and

(ii) a goal G** of the form $\square(Capacity_{ag} \geq X)$ requiring ag's capacity to a value smaller than or equal to X.



Limit goal load according to agent capacity: This tactic consists of refining the avoid goal (Goal Pattern 5.1) into a requirement or expectation stating that the agent capacity is greater than some value K, and a subgoal limiting the goal load value at any given time below or equals to K. Note the difference: while the previous tactic assumes a goal load and adapts the agent capacity accordingly, this one expects or requires an agent capacity and limits the goal load accordingly. This refinement pattern in shown in Tactic 5.6.

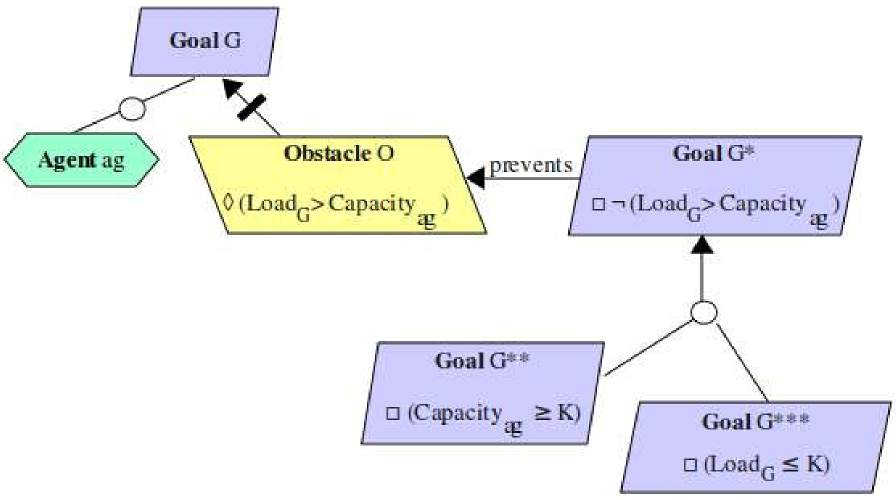
This tactic also has two specializations. On the specialization **limit goal load according to fixed agent capacity**, K is a constant value representing a fixed agent capacity. One typical strategy to satisfy this subgoal is to *limit the goal load over time*; for example, for a student enrollment system, this could be achieved by setting different registration dates for different student groups. The specialization **limit goal load according to varying agent capacity** is the dynamic counterpart of the previous specialization. The principle here is to monitor or predict variations in the agent capacity at run-time (i.e., the variation of k over time), and dynamically change the limit on the maximal goal load based on these variations. Satisfaction of goals that limit the load can be automated or left entirely under the responsibility of human agents.

Tactic 5.6: Limit goal load according to agent capacity

Postcondition: Define a new avoid goal G* of the form $\Box\neg(Load_G > Capacity_{ag})$ that is AND-refined into:

(i) a requirement or expectation G** of the form $\Box(Capacity_{ag} \geq K)$ requiring the agent capacity to be greater than or equal to some value K, and

(ii) a subgoal G*** of the form $\Box(Load_G \leq K)$ limiting the goal load to a value smaller than or equal to K.



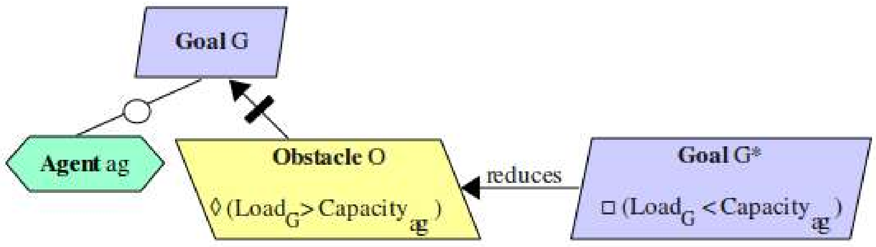
To illustrate some of these tactics with the IEF, consider the goal Avoid[Batch Size Exceeds Alert Generator Processing Speed] generated to prevent the obstacle Batch Size Exceeds Alert Generator Processing Speed. Applying the tactic 'set agent capacity upfront to worst-case load' refines this goal into a scaling assumption Batch Size Below Max (where Max is a constant value) and a subgoal Maintain [Alert Generator Processing Speed Above Max], i.e. the alert generator would have a fixed capacity designed to deal with the largest possible batch size. An alternative is to apply the tactic 'adapt agent capacity at runtime according to load', generating the subgoals Maintain [Accurate Batch Size Prediction] and Maintain [Alert Generator Processing Speed above Predicted Batch Size].

Obstacle Reduction

Instead of eliminating the scalability obstacle, as in the tactics above, one can reduce its likelihood by introducing a new goal containing some ad-hoc countermeasure. For scalability obstacles, one strategy to satisfy this new goal is to try to *influence the distribution of goal load*. Consider, for example, an e-commerce website wishing to announce a sale. It can send notification e-mails to registered customers on different dates, assuming that customers are more likely to shop as soon as they receive the e-mail. Note that this is different from *preventing* the obstacle occurrence by *limiting goal load over time*, where one sets an upper limit to the goal load at different points in time, as in the student enrollment example. In the e-commerce example, customers may still choose to buy in the very last day of the sale. Tactic 5.7 shows this resolution.

Tactic 5.7: Obstacle Reduction

Postcondition: Define a new goal $\texttt{G}^*$ containing some countermeasure that reduces the likelihood of the scalability obstacle O (e.g., $\texttt{G}^*$ with an objective function of the form $\textsf{Max P}(Load_G < Capacity_{ag})$).



Goal Weakening

Initial goals specification are often elaborated without considerations for the agent capacity. In this case, their obstruction by a scalability obstacle may be resolved by weakening the goal specification. The following elaboration tactics can be used:

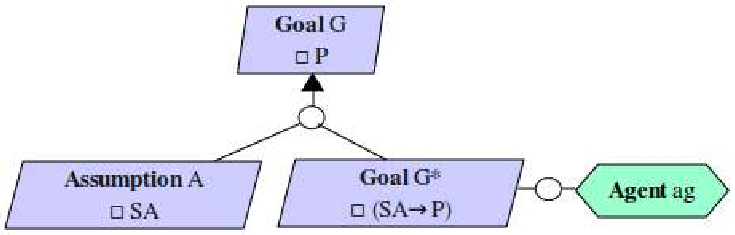
Introduce scaling assumption: Recall that the absence of a scaling assumption for a domain quantity means that there is no assumed constraint on its possible values. This tactic is applied when a goal of the form $\Box P$ is expected to handle certain domain quantities but does not refer to any scaling assumption, implying in a scalability obstacle. The tactic refines the obstructed goal into a scaling assumption and a subgoal whose definition states that the property expressed in the definition of the obstructed goal must be satisfied only when the property expressed in the scaling assumption holds. This tactics, therefore, determines which scaling assumptions to apply to a given goal: those who are concerned with domain quantities that are monitored or controlled by an agent in order to satisfy the goal. An example of this refinement pattern was given in the refinement of the goal Achieve [Batch Processed Quickly] (Goal 5.1) into the scaling assumption Expected Batch Size Evolution (Assumption 5.1) and the goal Achieve [Batch Processed Quickly Under Expected Batch Size Evolution] (Goal 5.2). Tactic 5.8 shows this refinement pattern.

Tactic 5.8: Introduce scaling assumption

Postcondition: The obstructed goal $\texttt{G}$ of the form $\Box P$ is refined into:

(i) a scaling assumption $\texttt{A}$ with a property $\Box\, SA$ defining the range of values the domain quantities can assume, and

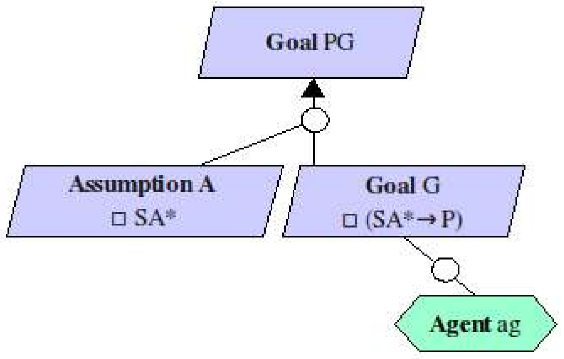
(ii) a subgoal $\texttt{G}^*$ of the form $\Box(SA \to P)$.



Strengthen scaling assumption: This tactic can be applied when a goal of the form $\Box(A \rightarrow P)$ cannot be satisfied because its responsible agent does not have sufficient capacity to support the load determined by its scaling assumption (See Table 5.1). This tactic strengthen the scaling assumption by reducing the range of values for its domain quantities. Tactic 5.9 gives its refinement pattern. Note that by strengthening the scaling assumption, one is simply saying that the domain quantities are believed to vary within a smaller range than previously stated. This is different from creating a requirement or expectation limiting the range of values domain quantities can assume, as in the tactic 'limit goal load according to agent capacity'.

Tactic 5.9: Strengthen scaling assumption

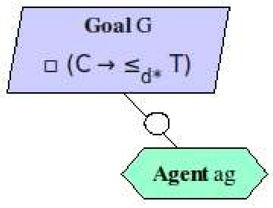
Postcondition: Replace the condition $\Box$`SA` in a scaling assumption `A` by another condition $\Box$ `SA`* describing a smaller range of values for the domain quantities in `A`.



Weaken goal objective function: This tactic involves weakening the goal definition, objective function or required levels of satisfaction so that it requires less agent capacity for its satisfaction. For Achieve goals of the form $\Box(C \rightarrow \Diamond_{\leq d} T)$, a specialized tactic for goal weakening is to **relax real-time requirement** by increasing the time for achieving the condition T. This formal refinement pattern is shown in Tactic 5.10. The **relax required level of satisfaction** specialization consists of relaxing the minimum value to be achieved for the goal objective function to be maximized (i.e. the value in the "Must" value of the goal's objective function).

Tactic 5.10: Relax Real-time Requirements

Postcondition: Replace the time d to achieve the target condition of the obstructed goal `G` by a larger value d^* so that the goal satisfaction can be achieved with the agent `ag`'s capacity.



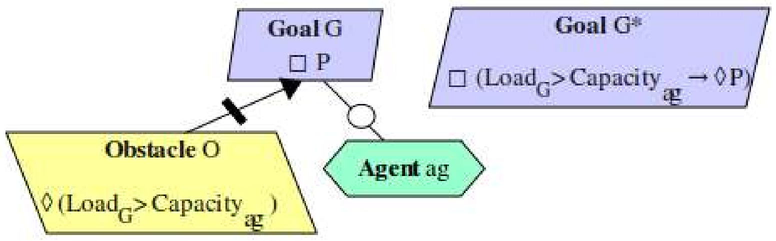
To illustrate some of these tactics, consider the first version of the goal Achieve [Batch Processed Quickly Under Expected Batch Size Evolution] (Goal 5.2), where this goal is specified with a fixed objective function. Assume that the scalability obstacle Expected Batch Size Evolution and Exceeds Alert Generator Processing Speed obstructing this goal is considered likely to occur. In order to resolve the obstacle, the tactic 'relax real-time requirement' would weaken the goal by increasing the expected processing time, the tactic 'relax required level of satisfaction' would weaken the goal by allowing for less than 90% of the batches to be processed in 8 hours.

Goal Restoration

It is generally impossible or too costly to guarantee that all scalability obstacles will be avoided. One should therefore also consider generating new goals to mitigate the effects of obstacle occurrences. This tactic tolerates the obstacle occurrence while requiring the target condition of the obstructed goal to be satisfied. For scalability obstacles, one strategy is to limit the load artificially as soon as it exceeds the agent capacity and submit the remaining load when the agent is less busy. Note that this is different from *preventing* the obstacle occurrence by 'limiting the goal load according to agent capacity'. As an example for the IEF, one can mitigate the occurrence of a scalability obstacle where the number of transactions that occurred in a day is exceptionally much larger than what had been anticipated (caused, for example, by a large scale security attack) by truncating the batch to a manageable size and reporting the unprocessed transactions to a later day when the number of transactions becomes manageable again. This refinement pattern is shown in Tactic 5.11.

Tactic 5.11: Goal Restoration

Postcondition: Add a goal G* that requires the target condition of the obstructed goal G to be eventually achieved.

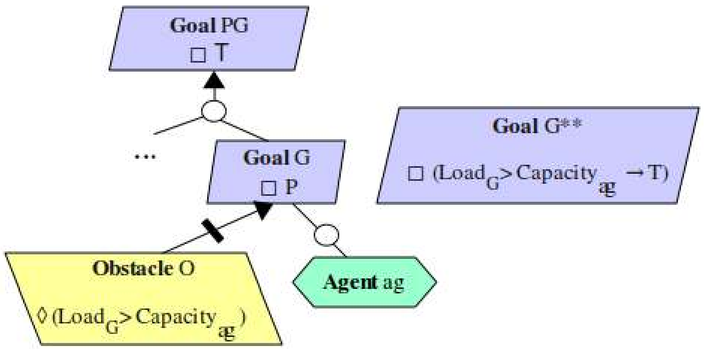


Obstacle Mitigation

In obstacle mitigation, a new goal is added to attenuate the consequences of an obstacle occurrence. That normally involves ensuring the satisfaction of a weaker version of the obstructed goal (weak mitigation) or of an ancestor of the obstructed goal (strong mitigation). Consider, for example, a call center with the goal Achieve [Calls Answered Quickly] stating that calls should be answered in less than 3 minutes. If in a given day, the call center receives a unusually high amount of calls, this goal is obstructed by the scalability obstacle Number of Calls Exceeds Customer Service Team Capacity. A strategy to mitigate this obstacle might be to create a new goal Achieve[Call Back Every Missed Call] which ensures a weaker version of the obstructed goal stating that a call must be taken in less than 3 minutes or the details of the customer should be recorded and the call returned in less than half an hour. The refinement pattern is shown by Tactics 5.12 and 5.13.

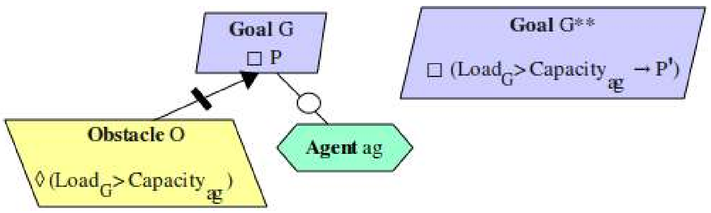
Tactic 5.12: Strong Obstacle Mitigation

Alternative postcondition: Add a goal `G**` that ensures the satisfaction of an ancestor of the obstructed goal `G` (e.g., the condition $\Box T$ of `G`'s parent goal).



Tactic 5.13: Weak Obstacle Mitigation

Postcondition: Add a goal `G*` that ensures the satisfaction of a weaker version of the obstructed goal `G` (e.g., a goal with a property $\Box P'$).



Generating Alternative Resolutions to Scalability Obstacles

We have used, throughout this section, the same IEF obstacle (Batch Size Exceeds Alert Generator Processing Speed) to exemplify a number of the resolution tactics presented in this chapter. These were *goal substitution*, *set agent capacity upfront to worst case load*, *adapt agent capacity at runtime*, *introduce scaling assumption*, *relax real-time requirement*, *relax required level of satisfaction*, and *goal restoration*. The application of two more tactics to reduce the likelihood of this obstacle are shown in Chapter 7, the *limit goal load according to agent capacity* and the *transfer goal to non-overloaded agent*. Our intent is to demonstrate the range of design alternatives that can be generated by systematic looking for instantiation of each resolution tactic. The set of resolution tactics, therefore, may help developers to consider a wider range of of strategies for solving a scalability obstacle than simply increasing the agent's capacity.

5.5 Critical Evaluation

The risks associated with the technique described in this chapter are as follows:

Risk: *Developers may feel that the costs of scalability requirements analysis outweighs its benefits.*

We present a technique for analyzing scalability at the goal level, but we have not yet investigated the costs of doing so.[9] Future research should include such an investigation. However, we have reasons to believe that the benefits brought by the technique outweighs the costs. As discussed in Chapter 1, we have observed that scalability problems often become apparent at the production phase, where the costs of fixing problems are much higher than they would be at the requirements engineering phase (Boehm, 1976). Analyzing scalability during requirements engineering also allows developers to consider a wider range of solutions to prevent scalability problems. Finally, this research assumes that KAOS is used for requirements engineering (see Chapter 1), meaning that we are simply adding another type of obstacle to be considered during the goal-obstacle analysis of the KAOS method, rather than suggesting that a complete goal model is built at a point of the development lifecycle where it may otherwise not be useful.

Risk: *It is difficult to estimate the agent capacity in early stages of the development lifecycle.*

A distinction must be made between the assumed capacity of given agents in the environment and the required capacity of the computer-based system to be constructed. The former is easier to estimate during requirements engineering time. However, the latter is typically unknown during requirements elaboration. In future work, we intend to develop techniques for identifying how the obstacles' likelihood varies with the required agent's capacity, as discussed in Chapters 7 and 8.

However, our technique for scalability goal-obstacle may be applied later in the development lifecycle to a pre-existing goal model, with the purpose of instantiating variables and functions for a quantitative scalability analysis. In this case, the capacity of the software agent might be known or be easier to estimate.

Risk: *Scalability obstacle assessment and resolution takes a simplified view of goal load and agent capacity.*

We have added two important concepts to KAOS: goal load (denoted as a domain-specific measure intended to characterize the amount of work needed to satisfy the goal) and agent capacity (denoted as a domain-specific measure intended to characterize the amount of resources available to the agent to satisfy the goal). These concepts, however, need to be elaborated further. For example, where and how to specify an agent capacity in a goal model? In the monitoring and control links? Can an agent have different kinds of capacity for different types of goal loads? Can there be conflicts between agent capacities that will make it difficult to offload, split, or transfer goals load? Also, defining the total load imposed by goals on an agent as the sum of the load imposed by the individual goals is a crude approximation. In reality, multiple goals will not all mobilize the agent's resource at the same time. For example, in the IEF, goals dealing with alerts generation mobilize the system's processing power during

[9] An idea of the effort involved can be taken from the Case Study 3, described in Chapter 7.

the night, whereas goals dealing with displaying the resulting alerts to fraud investigators mobilize the processing power during the day. Further elaborating the concepts of goal load and agent capacity, and developing more precise techniques for the assessment and resolution of scalability obstacles is subject of future research.

Risk: *Stakeholders may have diverging assumptions about how quantities in the application domain are expected to vary over time and/or across different categories of system instances.*

It is not uncommon to encounter diverging assumptions on the ranges of application domain quantities during the requirements engineering process. In the IEF, for example, diverging opinions could be explained by people's experiences with banks of different nature and sizes. Furthermore, the range of values for certain application domain quantities may depend on assumptions about other quantities. In the IEF, the expected number of transactions on a given day can be determined by an estimation of the number of accounts in that bank and the expected number of transactions per account per day. A technique for elaborating scalability requirements should offer systematic support for resolving diverging assumptions and assessing the combined effect that the variation of individual quantities have on dependent scaling assumptions and scalability goals.

In the IEF case study, we have used a parameterized goal model to highlight disagreements and show the impact of different assumptions on application domain quantities ranges in scaling assumptions and scalability goals. The parameterized goal model captured the dependency among scaling assumptions and scalability goals. Parameters represented the values variables can assume, and derived variables were described by refinement equations on these parameters. Instantiating parameters with different values highlighted divergences in scaling assumptions and allowed to compute the combined effect of these assumptions on scalability goals. Although this lightweight technique has helped in the requirements engineering process of the IEF, further research is needed in order to incorporate this technique into the scalability framework. For example, how to resolve the disagreements that were highlighted by the parameterized model?

Risk: *Considering the absence of a scaling assumption for a domain quantity as having an infinite range of values may be seen as imposing unnecessary work on the analyst.*

No system is infinitely scalable (Weinstock and Goodenough, 2006). Effectively, considering the absence of a scaling assumption as an infinite range of possible values for a domain quantity is forcing the definition of scaling assumptions for virtually every quality goal. This interpretation may seem counter-intuitive to those who believe that a requirements engineering technique should require minimum effort and accommodate incomplete information. While these principles are not incorrect, our conversations with computer scientists in both industry and academia suggest that scalability problems often occur because the range of values for some domain quantity and its effect on the system have been neglected. Systematically checking for the effect of scaling application domain variables in quality goals can uncover scalability problems that otherwise could be overlooked.

Contributions and Benefits

The contributions associated with the techniques described in this chapter are as follows:

Contribution: *A technique for describing, modeling and satisfying scalability requirements using KAOS.*

Our conversations with practitioners and the literature review suggest that scalability is largely overlooked during requirements engineering. A number of reasons contribute to this fact, such as time-to-market pressures and the lack of techniques to support the elaboration of scalability requirements (Industry Interviews, 2007). Furthermore, the multivariate nature of scalability, the difficulty in discovering domain quantity ranges at requirements engineering time, and the thin boundary between scalability requirements and design, make modeling scalability requirements a difficult task.

The technique we present in this chapter inherits from KAOS some advantages that are particularly useful for the elaboration of scalability requirements. It also extends KAOS to include the specification of scaling ranges in application domain characteristics and the varying degree of goal satisfaction over these ranges. Goal-obstacle analysis allows the systematic identification and resolution of potential scalability problems during the early stages of the software development. In particular, the technique helps to identify which scalability-related information needs to be elicited during requirements engineering: quality goals, scaling characteristics, goal load, agent capacity, and scalability obstacles. The technique also tackles some of the difficulties listed on Section 5.2. It is, for example, a step towards a taxonomy for domain assumptions as it specifies a new type of assumption: the scaling assumption, which allows for the description of time-varying assumptions on the application domain. Finally, the technique models scaling strategies in the goal tree as resolutions to scalability obstacles.

There are a number of research directions that can be taken in the future, such as: (1) using probabilistic information in a parameterized goal model, and investigating conflict resolution techniques to handle disagreements; (2) developing a context-sensitive agent load analysis that takes into consideration the time at which the load of a certain goal is imposed on an agent; and (3) further investigation into the relationship between cost goals and alternative obstacle resolution tactics, with the objective of investigating the cost-effectiveness of tactics.

Additional benefits of the technique introduced in this chapter are:

Benefit: *The technique offers a proactive approach to treat scalability problems caused exceptional application domain scenarios.*

Our conversations with practitioners and researchers suggests that scalability problems are, on occasion, caused by exceptional application domain scenarios that are difficult to predict. The technique addresses the problem of scalability requirements through a cycle of identification, assessment and resolution of scalability obstacles. By applying a goal-obstacle analysis, our technique takes a pessimistic view of the goal model developed so far, allowing for the natural investigation of scalability problems caused by exceptional circumstances, which could be overlooked easily if no systematic technique was being used.

Benefit: *A goal model can serve as input to various methods for analyzing software scalability at the architectural level.*

The correctness and usefulness of quantitative methods for assessing system qualities is highly dependent on the variables and functions selected for the analyses. In the scalability framework (Figure 4.1), these variables and functions can be systematically derived from the goal model elaborated using our requirement engineering technique. Theoretically, this goal model may also be used as input to various methods of analyzing software scalability at the architectural level (Kazman et al., 1998, 2001; Bahsoon and Emmerich, 2008). Instantiation rules would have to be developed for different analyses. Further research and case studies are needed to develop such rules.

Benefit: *A goal model can have different uses during the development lifecycle.*

The goal model with carefully specified scaling assumptions and scalability goals can be used in various forms throughout the software development lifecycle. During requirements engineering time, it can guide requirements engineering decisions. In the design phase, it can be used to guide design-level decisions, such as choosing between some backbone algorithms, architectures or technologies. At testing time, it may serve as input to models that predict performance for different hardware infrastructures. We are particularly interested in developing a technique for designing and selecting scalability test cases from goal models.

5.6 Summary

This chapter has described a systematic approach for specifying and reasoning about scalability during a goal-oriented requirements elaboration. The approach involves the precise specification of scaling assumptions and scalability goals. The former are assumptions about expected variations of quantities in the application domain. The latter are goals whose specification includes a definition of the required levels of goal satisfaction under the scaling of domain characteristics specified in the scaling assumptions. The technique explores scalability at the goal level, by systematically identifying, assessing, and resolving obstacles to the satisfaction of scalability goals. The resolution of scalability obstacles at this level allows one to explore a wider range of alternative resolutions than at the design or implementation levels, notably through the exploration of alternative goal refinements and alternative assignments of goals to agents.

Now that the different techniques of our framework have been presented, we next describe how they fit together in a method for characterizing and analyzing the scalability of software systems.

Chapter 6

Scalability Analysis Method

The techniques described in the previous chapters are combined in a method for the analysis and characterization of software system scalability. In this method, the variables and functions used in the scalability analysis are systematically derived from the system's goal model. The method can be applied at any point of the development lifecycle, bringing greater benefits if adopted earlier, when developers have more freedom to negotiate requirements and change the system design.

6.1 The Method Description

This now describe a first attempt to define a method that combines the techniques described in the previous chapters. The method defines the set of activities required to produce the elements of the scalability framework (Figure 4.1) and to perform a scalability analysis. Our objective is to provide the necessary structure and steps to derive, from the requirements elaboration technique, the variables and functions to be used in the scalability analysis. The method is illustrated by Figure 6.1.

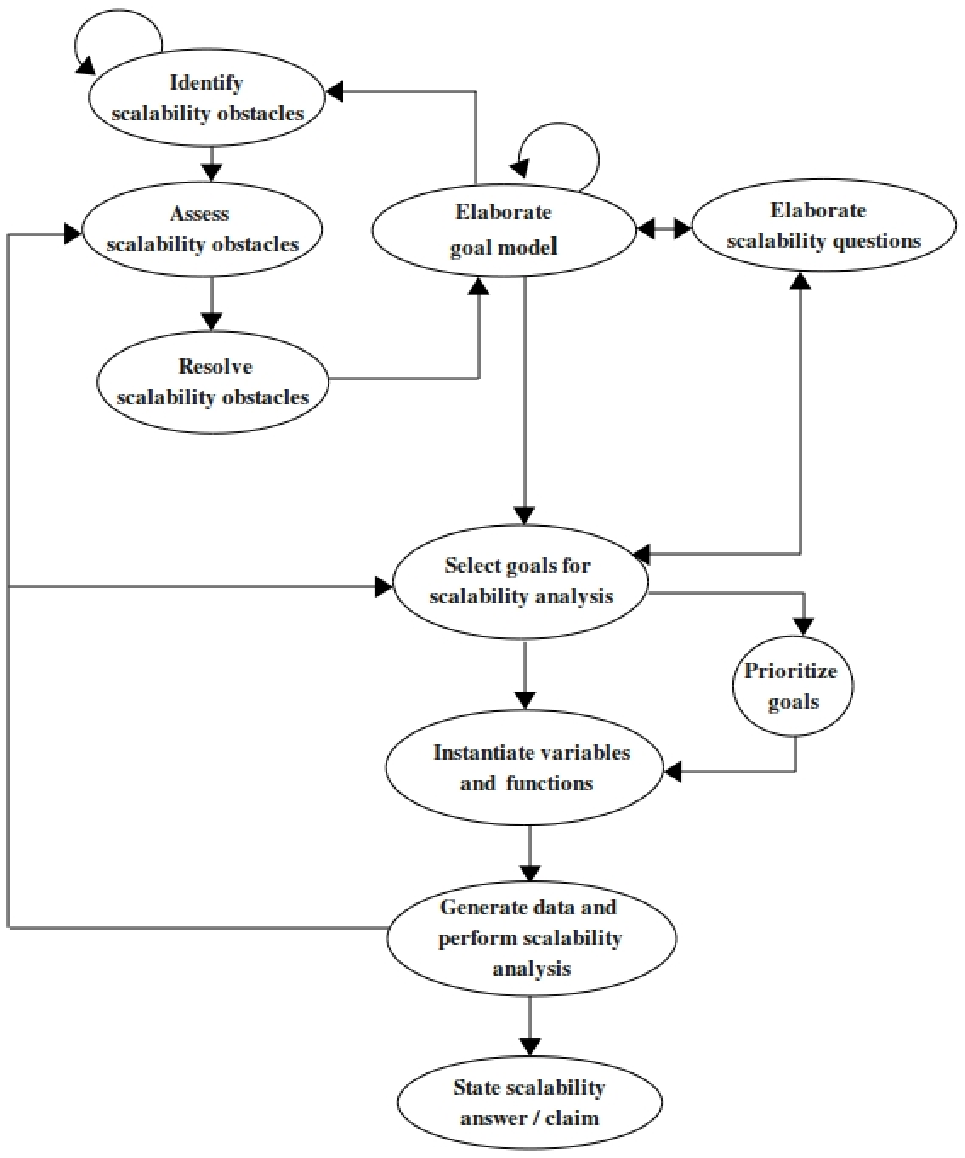

Identify scalability obstacles
Assess scalability obstacles
Resolve scalability obstacles
Elaborate goal model
Elaborate scalability questions
Select goals for scalability analysis
Prioritize goals
Instantiate variables and functions
Generate data and perform scalability analysis
State scalability answer / claim

Figure 6.1: Overview of the scalability analysis method.

In summary, the objective of a scalability analysis is to answer a scalability question. That requires a good understanding of the system's scalability goals and the characteristics of the application domain and system design that will influence these goals. Such understanding starts with a conventional KAOS goal model, from where scalability obstacles are iteratively identified, assessed and resolved, resulting in an updated model that includes scaling assumptions and scalability goals. Selected goals are then used to derive the variables and functions required for the analysis. Finally, the analysis is conducted and the answer to the scalability question stated. We next describe each of these steps.

6.1.1 Define the Scalability Question

Initial scalability questions may originate from various sources. For example, questions may refer to recurring causes of scalability problems in similar systems, such as the questions listed in Weinstock and Goodenough's scalability audit (Weinstock and Goodenough, 2006). This audit is targeted at systems that are required to accommodate a significant increase in load over their lifetimes or whose failure caused by an unexpectedly high workload would be disastrous. The audit covers potential bottlenecks, common incorrect assumptions, general scaling strategies, and a scalability assurance method. It is, therefore, a good source of inspiration for identifying potential scalability problems; however, being described in only six pages, the audit is also very limited. Another possible source for initial scalability questions may be a impact analysis of planned software changes, exceptional application domain scenarios, or unusual system usage patterns. Alternatively, scalability questions simply may come from examining the system's conventional KAOS goal model—hence, the directed link from the step "elaborate goal model" to the step "elaborate scalability question". In the IEF, for example, an initial scalability question may be stated as: "Is the IEF scalable with respect to processing time when the number of records in a batch vary over time?" This is an example of a typical early stage question, when not much information is known about the scaling of the system's application domain, its design, or its (possibly competing) scalability goals.

The initial scalability question will inform what portions of the goal model need to be elaborated; that is, for which portion one needs to identify relevant scalability goals and scaling assumptions that will yield the variables and functions to be used in the framework. Hence the directed link from the step "elaborate scalability question to the step "elaborate goal model" in Figure 6.1.

6.1.2 Elaborate the Goal Model

In order to analyze scalability, one should seek to improve his/hers knowledge about the system, its environment, and its scalability goals. The first step towards this objective is to build, or refer to, a conventional KAOS goal model. By "conventional" we mean a goal model that has been built without scalability in mind; that is, a model that ordinarily would be produced by the KAOS method, described in Lamsweerde's book (van Lamsweerde, 2008). This is the step "elaborate goal model" in Figure 6.1. [1]

In the IEF, the goal related to the initial scalability question is a performance goal named Achieve[Batch Processed Quickly] (Goal 5.1). To ease the discussion, we re-state this goal below:

[1]In this thesis, KAOS is assumed as the technique for requirements engineering (see Chapter 1). Therefore, the analyst either has access to a existing goal model or will build one during the requirements engineering phase.

Goal 6.1

Goal Achieve [Batch Processed Quickly]

Category Performance

Def Daily batches of transactions provided by the bank should be processed in less than 8 hours.

Quality Variable: processingTime: Time

Def: The time required to process the batch

Sample Space: Set of batches submitted.

Objective Functions: At least 9 out of 10 batches should be processed within 8 hours, and all batches should be processed within 9.6 hours.

Name	Definition	Mode	Must
% of Batches Processed in 8 hours	Pr(processingTime $\leq$ 8h)	Max	90%
% of Batches Processed in 9.6 hours	Pr(processingTime $\leq$ 9.6h)	Max	100%

6.1.3 Identify, Assess and Resolve Scalability Obstacles

The purpose of a scalability goal-obstacle analysis is to verify whether system agents in the goal model have sufficient capacity to handle the varying loads imposed on them. As a result, the model is extended to describe how quantities in the application domain are expected to vary over time and/or across different categories of system instances (scaling assumptions), and the required levels of quality goal satisfaction under the variations of these application domain quantities (scalability goals).

Starting from the conventional KAOS goal model, a scalability goal-obstacle analysis consists of systematically (1) identifying scalability obstacles that may obstruct the satisfaction of the quality goals, requirements, and expectations; (2) assessing the likelihood and criticality of these obstacles; and (3) resolving them by modifying existing goals, requirements, and expectations, or generating new ones so as to prevent, reduce, or mitigate the identified obstacles. The process is represented in Figure 6.1 by the steps "identify scalability obstacles, "assess scalability obstacles" and "resolve scalability obstacles", which will lead back to the step "elaborate goal model". This process is iterative.

In Chapter 5, the performance goal Achieve[Batch Processed Quickly] was refined to consider the scaling of the transactional batch size (Assumption 5.1). In reality, however, the batch processing time also depends on the number of distinct business entities in a batch, the number of fraudulent patterns that an incoming transaction is compared against when entering the system, and the type of the fraudulent patterns (e.g., historical, peer, or blacklist).[2] We, therefore, refine Goal 6.1 into the scalability goal Achieve[Batch Processed Quickly Under Scaling Assumptions] (Goal 6.2) and the scaling assumptions Expected Batch Size Evolution, Expected Number of Business Entities Variation and Expected Number of Fraudulent Patterns Variation. This goal is shown by Goal 6.2 below. The type of a fraudulent pattern is represented in a goal model as a standard KAOS domain assumption (we will refer to these as *non-scaling domain assumptions*).

[2]In the IEF, one of the techniques used to identify fraudulent transactions is to compare all transactions against known fraudulent patterns. Some types of patterns require more complex processing than others . Real classification of patterns types have been omitted to protect Searchspace's intellectual property.

Goal 6.2

Goal Achieve [Batch Processed Quickly Under Scaling Assumptions]
Category Performance and Scalability
Definition Daily batches of transactions provided by the bank source system should be processed in less than 8 hours, provided that the batch size does not exceed the bounds stated in the scaling assumptions "Expected Batch Size Evolution", "Expected Number of Business Entities Variation" and "Expected Number of Fraudulent Patterns Variation".
Quality Variable processingTime: Time
Objective Functions Under the conditions stated in the scaling assumptions "Expected Batch Size Evolution", "Expected Number of Business Entities Variation" and "Expected Number of Fraudulent Patterns Variation", for any bank, at least 90% of the daily batches should be processed within 8 hours, and all batches should be processed within 9.6 hours.
Responsibility Alert Generator

Assessment and resolution of a scalability obstacle may require a more precise, quantitative assessments of an agent's ability to satisfy a given load. Whether capacity can be estimated will depend on the agent and the phase of the development lifecycle in which the system finds itself. It is interesting to note that assessing a scalability obstacle effectively may be answering a particular scalability question about the agent's ability to satisfy its scalability goals. Such a question may be answered in the step "generate raw data and perform scalability analysis", hence the data dependency arrow between this steps and the "assess scalability obstacle" in Figure 6.1.

As the scalability goal-obstacle analysis progresses, the analyst updates the goal model, adding scaling assumptions, refining quality goals to describe their required level of satisfaction under variations in the scaling assumptions and, possibly, creating new goals. By the end of it, the analyst will have acquired more knowledge about the stakeholder needs and expectations, physical and financial constraints on the system, exceptional circumstances, characteristics of the application domain and expected changes to these characteristics in the future. In particular, the scalability goal-obstacle analysis is likely to have uncovered exceptional application domain scenarios and unusual patterns of system usage that may impose scalability risks.

6.1.4 Select Goals for Scalability Analysis

At first sight, one can draw an almost direct parallel between KAOS quality variables and objective functions and the variables and functions in the scalability framework. Nevertheless, this instantiation is not so straightforward. The goal model will contain a number of goals, assumptions and agents, many of which are not relevant to the scalability concern at hand. Therefore, relevant goals should be selected before refining the scalability question and deriving the analysis variables and functions. An obvious first step is to consider the scalability goals and scaling assumptions in the model. These, however, will be yet too many. In practice, the final selection of goals depends on the objective of the scalability analysis. We next discuss initial ideas for performing the step "select goals for scalability analysis" in Figure 6.1 for three types of scalability concerns:

Understanding the scaling trend of a particular goal in isolation: The simplest form of analysis seeks to characterize the scaling trend of a goal independently of other goals. That includes both the effect of scaling application domain characteristics and the effect of variation in some system design characteristics. This kind of analysis is useful, for example, to understand the demand of a particular goal on an agent and to test the effectiveness of a scalability strategy to maintain or improve the satisfaction of a specific goal. In such cases, the goals selected for the analysis should be (1) the scalability goal whose trend should be characterized; (2) all scaling and non-scaling domain assumptions related to that goal; and (3) all agents assigned to that goal.

Analyzing an agent's ability to support the load imposed on it: Another form of analysis may investigate an agent's ability to satisfy a given quality goal for a range of application domain characteristics. The satisfaction of a goal cannot be measured in isolation. An agent is often assigned to multiple goals, among which the agent's capacity must be shared. Therefore, testing the satisfaction of a goal requires that all other goals sharing resources with that goal are also being satisfied. Such an analysis requires therefore the identification of all goals that might simultaneously impose load on that particular agent. One way to do so is to generate an agent load diagram (van Lamsweerde, 2008) and use informal techniques, such as experts' knowledge, to identify goals that will impose load simultaneously. [3] In this case, the goals selected for analysis should be (1) the scalability goal we are interested in; (2) the agent whose ability to satisfy that goal we aim to analyze; (3) all goals that might also require the resources of that agent; (4) the scaling and non-scaling domain assumptions associated with all selected goals; and (5) possibly some cost goal if we want to investigate alternative scalability strategies to satisfy that goal.

Characterizing the trade-off between a set of competing goals: One may be also be interested in analyzing how the satisfaction of a particular scalability goal (or a particular strategy to achieve scalability) may affect the satisfaction of other goals. Such analysis is not only useful when the goals share the resources of a same agent. The IEF, for example, compares incoming transactions against fraudulent patterns to identify possibly fraudulent transactions. One alternative to support a greater number of transactions within a time window (performance and scalability goal) is to use simpler patterns, such as patterns that do not use historical information. Such a tactic improves the processing time, but it might have a negative effect on the goals Achieve[Reduced Rate of False Alerts] and Achieve[Reduced Rate of Missed Frauds]. If we want to take such trade-offs into consideration when analyzing scalability, the goals selected should be (1) the scalability goal we are interested in; (2) other quality goals whose satisfaction may be affected by the satisfaction of the scalability goal of interest; (3) the scaling and non-scaling domain assumptions associated with all involved goals; and (4) the agents responsible for all the involved goals.

Examples of goals selection according to the above guidelines are given in Chapter 7. However, more research is needed to develop goal selection techniques further for different types of scalability concerns.

[3] Formal techniques for identifying concurrent load are subject of future work.

6.1.5 Refine and Elaborate New Scalability Questions

The scalability question will inform which goals need to be selected for the scalability analysis. Conversely, the selection of goals may also help the *systematic* refinement of scalability questions. Hence the bi-directed arrow between the steps "elaborate scalability question" and "select goals for scalability analysis". A question typically will focus on a particular scalability aspect of interest—involving one or a few *scalability goals*. The scaling characteristics of the application domain (and their ranges) are derived from the *scaling assumptions* associated with these goals, while the scaling characteristics of the system design come from the *agents* responsible for these goals. The scalability question may also explicitly refer to the *quality variables* and *objective functions* of the scalability goals of interest.

For example, the question "Is the IEF scalable with respect to processing time when the number of records in a batch vary over time?" (Section 6.1.1) is related to the goal Achieve[Batch Processed Quickly Under Scaling Assumptions]. This goal is assigned to agent Alert Generator and refers to the scaling assumptions Expected Batch Size Evolution, Expected Number of Business Entities Variation, and Expected Number of Fraudulent Patterns Variation. For the purpose of illustration, assume that the Alert Generator is responsible only for this goal. According to the scaling assumptions, the processing time depends not only on the number of transactions in the batch (which may reach 95 million), but also on the number of business entities (up to 60 million) and the number of fraudulent patterns used to identify fraudulent transactions (up to one thousand). The definition of the goal Achieve[Batch Processed Quickly] also states that not all batches need to be processed in 8 hours and that there is actually a 20% tolerance for 10% of the batches. The original question can then be refined to "Can the IEF process 90% of the batches within 8 hours and all batches within 9.6 hours, when the number of transactions in a batch scales up to 95 million, the number of business entities scale up to 60 million, and the number of fraudulent patterns scale up to 1000?".

The elaboration of the IEF goal model also uncovered requirements related to the storage of transactions, fraudulent patterns, and alerts. These new requirements led to the definition of new scalability questions concerned with the system's ability to store this data.

6.1.6 Instantiate Variables and Functions for the Scalability Analysis

The step "instantiate variables and functions" in Figure 6.1 is similar to the process for refining, or defining new, scalability questions. It, however, may involve more goals, depending on the scalability question, as is shown below.

Map Goal Variables to Analysis Variables

The variables required for the scalability analysis are as follows:

- **Independent variables** represent aspects of the application domain and system design that will affect the system behavior. They may vary over a range or be set to fixed values. Scaling independent variables belonging to the application domain are derived from *scaling assumptions*. More precisely, they correspond to the domain quantities whose possible range of values are determined in the scaling assumptions. Note that this range of values also represents the *load imposed on goals* by these assumptions. Non-scaling independent variables belonging to the application domain are derived from conventional KAOS domain assumptions.

System design variables correspond to characteristics of the software, devices, and humans composing the system, which are derived from characteristics of the *agents responsible for the scalability goal*—that includes characteristics that will determine the *agent capacity* (e.g., disk size or number of processors). When the agent is the software, instantiating these variables may require knowledge of the portion of the system design (or intended design) that satisfies that goal. This information might be available depending on in which point of the development lifecycle the system is when the scalability analysis is performed.

- **Dependent variables** represent software metrics that characterize quality goals whose specification describe the required level of satisfaction under the variation of independent variables. Therefore, dependent variables are represented in the goal model as *quality variables of scalability goals*.

  Furthermore, scalability problems are often caused by resource exhaustion. Sometimes, there are business reasons to set the target levels for resource usage under the variation of application domain quantities (such as when the system is deployed on a hosted environment). In such cases, these targets are represented in scalability goals. However, on many other occasions, the only concern is not to exhaust the system's resources—a concern that is not usually modeled as explicit goals, but that is taken into consideration in virtually all scalability analyses. Therefore, there will likely be dependent variables that measure resource usage in a scalability analysis. In other words, dependent variables are also derived from *characteristics that measure the usage of an agent's capacity*.

- **Nuisance variables** are characteristics of the application domain and the system design that cannot be manipulated in a scalability analysis. They are derived from goal models in the same way as independent variables.

As an example, take the scalability goal Achieve[Batch Processed Quickly Under Scaling Assumptions] (Goal 6.2), assigned to the agent Alert Generator. This goal describes the acceptable performance for batch processing in the IEF and is associated with the scaling assumptions Expected Batch Size Evolution, Expected Number of Business Entities Variation and Expected Number of Fraudulent Patterns Variation. Searchspace intends to achieve this goal by distributing the load over a cluster of multi-threaded processing engines, with a number of engines per computer. Therefore, a possible instantiation for the scalability analysis variables is as follows:

Independent variables:

Application Domain:

Number of transactions batch [10,000 — 95 million]

Number of business entities in a batch [6,000 — 60 million]

Number of fraudulent patterns [200 — 1000]

Type of fraudulent patterns [historical, peer, blacklist]

System design:

Number of processing engines in the cluster

Number of threads within a processing engine

Network bandwidth connecting the cluster

Disk read/write speed

CPU speed

Dependent variables:

Average processing time

Average CPU utilization

Maximum memory usage

Maximum disk usage

Total network I/O time

Total disk I/O time

Note the source of each variable: the number of transactions, entities and fraudulent patterns were mentioned in the scaling assumptions Expected Batch Size Variation, Expected Number of Business Entities Variation, and Expected Number of Fraudulent Patterns Variation, respectively. The type of fraudulent pattern is taken from a conventional KAOS domain assumption named Types of Fraudulent Patterns. The number of processing engines and threads, network bandwidth, disk speed and CPU speed are characteristics of the agent Alert Generator. The processing time came from the variable in the scalability goal Achieve[Batch Processed Quickly Under Scaling Assumptions. Finally,s CPU usage, memory usage, disk usage, network I/O time and disk I/O time came from characteristics that determine Alert Generator's capacity usage.

From Objective Functions to Preferences

Preference functions measure the "satisfaction level" of the stakeholder with respect to individual quality goals. They should be modeled in such a way that the values of dependent variables are mapped to a range of preference values in which a greater value equates with greater preference. Preferences are derived in the goal model from *objective functions in scalability goals*.

When translating from objective functions to preferences, one should avoid creating a "ranking" between system designs that equally meet the target defined by the objective function. For example, if an objective function states that a system should process a data batch in less than 8 hours, then a stakeholder might be tempted to assign a greater preference to a system that processes all batches within 6 hours than one that does so within 8 hours. In our opinion, such assignment is arbitrary if it cannot be justified in terms of the system's business goals and may impose an unnecessary demand on the system design. However, if the objective function states that the system should *maximize* the probability that processing will be completed in less than 8 hours for 90% of the batches, then the preference function can be modeled in such a way that preference value assigned to a system increases with the percentage of batches that are processed within 8 hours. For illustration, consider the preference function derived from the scalability goal Achieve[Batch Processed Quickly Under Scaling Assumptions] (Goal 6.2):

$$\text{pref(ProcTime)} = \begin{cases} -10.000, & \text{if Pr(ProcTime} \leq \text{8hrs)} < 0.9 \text{ or Pr(ProcTime} > \text{9.6hrs)} > 0 \\ \dfrac{\text{Pr(ProcTime} \leq \text{8hrs) - 0.9}}{0.1}, & \text{otherwise.} \end{cases} \tag{6.1}$$

An analyst may as well choose to model preferences on some resource usage even when there are no explicit goals associated with that resource. Such a preference can be used to show scaling trends that may lead to resource exhaustion. However, care must be taken not to base decisions on preferences if there are no business reasons behind them. One way to do that is not to include such preference is in the calculation of the system utility.

Further research is required to devise other guidelines for translating from goal objective functions to preference functions.

From Goal Prioritization to Utility Functions

A **utility function** transforms a vector of preference values into a single scalar value. There are different ways of modeling a utility function. In Chapter 4, we used objective weighting. A disadvantage of this approach is that direct estimation of weights is subjective and vulnerable to judgment errors. A more precise and consistent set of weights can be determined by applying a requirements prioritization technique. Some of these techniques produce a list of requirements and their respective weights, which is used as a conceptual map for analyzing and discussing the candidate requirements. In the scalability analysis, preference functions can be mapped back to goals. Therefore, the weights of preferences can be inherited from the weights assigned their originating goals by a requirements prioritization technique. This is the step "prioritize goals" in Figure 6.1.

There are a number of prioritization techniques available, the Analytic Hierarchy Process (AHP) being a well known one (Saaty, 1980). AHP performs pairwise comparison of requirements, which includes redundancy and is thus less susceptible to judgmental errors. A known disadvantage of pairwise comparison is that such approaches become impractical when the size of the collection of requirements is greater than about twenty, since the elicitation effort grows as the square of the number of requirements (Avesani et al., 2005). A scalability analysis, however, is likely to focus on a specific scalability concern and is unlikely to involve that many goals. Further research is needed in order to evaluate the use of different requirements prioritization techniques to derive a utility function.

6.1.7 Perform the Analysis and State the Answer

With the scalability question translated into clearly defined variables and functions, the analyst should proceed to the step "generate raw data and perform scalability analysis" in Figure 6.1. The analyst first needs to produce the raw data for the analysis; that is, all the values assumed by the dependent variables in a scalability experiment. In this research, we have used testing for that purpose. However, we believe it to be possible to generate the raw data through modeling or a combination of modeling and testing, as it will be discussed in Chapter 8. Such activity is not trivial. The scalability experiment must be carefully designed, considering the adequacy of the chosen technique to the problem at hand. When using testing, for example, the analyst should consider whether a running version of the system is available, the effort of building a prototype, the power of the available infrastructure, and so forth. If using a queuing network, the analyst must verify whether the system can be abstracted into servers and queues, which type of queue better models the system, whether it is possible to estimate the probability of transitions between service centers, job arrival and service times, and so forth. In the IEF case study, testing was used to generate the raw data. A detailed description of the study is given in Chapter 7.

Once the raw data has been generated, it is used to calculate the value of preferences and utility in the scalability analysis, to then provide an answer to the scalability questions. The scalability answer should be very carefully stated, with sufficient information to enable the reader to confirm or dispute the claim. At the very least, the answer should state the scalability goal or trend being verified, the characteristics of the application domain and system design considered in the analysis, and the scaling range of values these variables are expected to assume. For example, an answer to the scalability question stated in Section 6.1.4 could be: "The IEF is scalable with respect to response time because it can process batches of 10,000 to 95 million transactions against a data store of up to 60 million business entities and up to 1000 fraudulent patterns in less than 8 hours in 92% of the cases and no longer than 9.6 hours for all cases, by varying the number of processing engines and machines in the cluster. That is, the system satisfies the scalability goal Achieve[Batch Processed Quickly Under Scaling Assumptions]".[4] In this case, the system is considered scalable *with respect to response time* because it satisfies the goal to which the scalability question refers. The scalability answer, however, does not need to be in the form of a "yes" or a "no", nor does it need to be presented exclusively in textual form. Instead, it may include as much information as considered necessary. For example, it could show graphs representing the trends on the satisfaction of the quality goals against the different scaling characteristics of the application domain and system design. Its format will thus depend on the scalability question. Future research should look into the information required to answer different types of scalability questions satisfactorily. This corresponds to the last step of the method, the "state scalability answers/claims" in Figure 6.1.

6.2 Critical Evaluation

The risks associated with this chapter are as follows:

Risk: *The model may lose accuracy when translating from objective functions to preferences.*

Letier and Lamsweerde suggest specifying goal objectives in terms of probabilistic functions (Letier and van Lamsweerde, 2004). An important advantage of describing quality goals with probabilistic objective functions is that the goal satisfaction criteria can be based on domain-specific physical interpretation, rather than subjective judgment. Converting objective functions into preferences

[4]If a cost goal has also been considered in the analysis, the claim could be extended to say "for a total cost of 'x' per transaction, satisfying the goal Reduce[Total Cost of Ownership]".

and utility functions makes room for criteria that have no physical meaning. If care is not taken, the accuracy introduced by the goal model may be lost in the scalability analysis. This is undoubtedly a disadvantage of such transformation and future research directions should develop transformation rules that will guarantee that the physical meaning of objective functions is not lost.

Risk: *Using objective weighting to model the utility requires variables to be independent.*

There are a number of ways of modeling a utility function. In the scalability framework, as in ATAM and CBAM, we have used objective weighting. This approach can only be used if the variables involved are independent. In our framework, the variables used in the objective weighting are derived from quality variables in a goal model. Goals, in KAOS, can be refined until they are independent; this, however, does not guarantee the independence of their quality variables. When quality variables cannot be assumed independent, the variables derived from them need to be defined further until other variables are reached that can be assumed to be independent.[5] More research is needed into refinement techniques that will convert goal quality variables into variables that can be used in a weighted sum. Alternatively, other forms of modeling a utility function can be investigated.

Risk: *There is a gap between the instantiation of analysis variables/functions and the design of the scalability experiment.*

The method, as stated, does not define how to select a small, yet representative, set of experiments to characterize the scalability of a system. Furthermore, the method requires the analyst to have sufficient knowledge and experience to design experiments that will provide the correct raw data required for the analysis.

In this research, assumptions guarantee that testing can be used to generate raw data and that the metrics collected are a fair representation of the system qualities of interest. Designing the scalability experiment is one of the main areas of future work. We intend to integrate widely used models, such as queuing networks and Petri Nets, into the scalability framework and to provide guidelines to design and select test cases for scalability.

Risk: *Some of the steps described in the method are just initial ideas that have not been validated.*

The method described in this chapter brings together techniques for scalability analysis and for the elaboration of scalability requirements. Each of these techniques have been validated separately, as will be discussed in Chapter 7. The whole method, however, has not. Some steps of the method are just initial ideas; when that is the case, we have made a point of clearly stating that more research is needed to develop them.

Contributions and Benefits

The contributions associated with this chapter are as follows:

Contribution: *A <u>method</u> for characterizing and analyzing software system scalability that is independent of the application domain and system design.*

[5]This assumption is also made for quantitative analysis of fault trees and for the computation of Bayesian networks.

The method introduced in this chapter combines a technique for elaborating scalability requirements with a technique to characterize and analyze software system scalability. The techniques complement each other in the sense that the variables and functions used in the analysis are derived from the goal model produced by the requirements engineering technique, and scalability obstacles identified in the goal model requiring a more thorough assessment can be supported by the analysis technique. These two main components rely on concepts that are common to software systems in general, such as agents, quality goals and software metrics, producing a method that can be used across application domains and system designs. Yet, the requirements engineering technique ensures that the analysis will be relevant to the problem at hand.

An additional benefit of the method introduced in this chapter is:

Benefit: *A method that encourages a proactive approach to scalability when building systems.*

As observed in Chapter 1, developers should take a proactive approach to scalability, designing systems that will scale to meet their goals without running up against economical or fundamental physical limits.

The method described in this chapter naturally fits the development lifecycle, encouraging a proactive approach to scalability. During the requirements engineering phase, scalability obstacles can be identified, assessed and resolved, providing the developer with more freedom to investigate alternative system designs. Obstacles that cannot be assessed during the requirements engineering can be marked for later investigation. As the design phase advances and the required information becomes available, these obstacles can be revisited and the scalability analysis performed.

6.3 Summary

This chapter combines the techniques described previously, composing a method for the analysis and characterization of software system scalability. The method is as follows: starting from an initial scalability question, the analyst updates a goal model of system requirements through the identification, assessment and resolution of scalability obstacles. Selected scaling assumptions and scalability goals are used to refine the scalability question and to derive variables and functions for the scalability analysis. The result is an answer to the scalability question. The application of the method can start at any point of the development lifecycle, bringing greater benefits if adopted earlier, when developers have more freedom to change the system design and negotiate requirements.

Chapter 7

Empirical Studies and Examples

This chapter describes case studies with a large industrial system and a number of other smaller examples used to demonstrate the concepts and techniques described in this thesis. The empirical studies include two case studies to validate the scalability analysis technique and a requirements engineering exercise using our extensions to KAOS for elaborating scalability requirements.

7.1 On the Evaluation of Techniques

This chapter describes the three case studies completed during the course of this PhD research. Case studies were chosen as the method of validation for the concepts and techniques described in this thesis for the following reasons: (1) we were interested in investigating the suitability of the concepts and techniques in a typical industrial project; (2) we were fortunate to have full access to Searchspace's code base, documents and employees; and (3) the alternative method usually used to validate novel techniques—a formal experiment—required careful control, statistical rigor, and appropriate levels of replicability, which would be too difficult to design for a large, proprietary and complex software system. Case studies, however, are harder to interpret and more difficult to generalize (Kitchenham et al., 1995). The limitation of our case studies are discussed in the end of each case study.

The subject of the studies was the *Intelligent Enterprise Framework (IEF)*, a real-world data analysis system for the financial services industry. A short description of the system is given before introducing the case studies. This chapter also presents a number of examples taken from the computing literature and from industry interviews to demonstrate the use of the scalability framework across different application domains and system designs.

Two retrospective empirical studies were carried out to demonstrate that our scalability analysis technique provides the information required to characterize, analyze and compare the scalability of software systems according to our definition of scalability; that is, with respect to the characteristics of the application domain and system design that are expected to vary, and the levels of satisfaction of quality goals measured under these variations. For such, it is assumed that the acceptability and utility of this definition have been convincingly argued through discussions in Chapters 1 to 3 and comparisons with other definitions in the computing literature.

The first study was designed to provide an early validation of the analysis technique and give quick feedback for the research. More precisely, it consisted of a small scale empirical study that compared prototypes of two consecutive IEF versions. The second study repeated the study for the real IEF. The third study applied the requirements engineering techniques described in Chapter 5 to elaborate the scalability requirements of a new version of the IEF.

The structure of the case studies are adapted from (Kitchenham et al., 1995) and have the following sections: (1) objectives, (2) hypothesis, (3) pilot project, (4) planning, (5) execution, (6) evaluation, (7) concluding remarks.

7.2 The Intelligent Enterprise Framework (IEF)

As described in Chapter 1, the IEF is a data analysis system for the financial services industry. The system receives large volumes of financial transactions data, builds adaptive profiles of business entities within the data, and generates alerts to automatically notify users of behavior that appears unusual. Searchspace's users are primarily the staff of retail, investment banks and other financial institutions (we use the word *banks* to consider this group). Its data analysis comprises the following stages: validation, preprocessing, loading, migration, profiling, eventing and alerting. The first stages of the data analysis are performed by a sub-system, the *Data Manager*, which is responsible for the validation, preprocessing, loading and migration of the transactional data to a database. In the preprocessing stage, the Data Manager takes transactions as input and replaces the various original business entity identifiers with simplified *surrogate keys*. Normally, a transaction contains a heterogeneous mixture of identifiers for different kinds of business entities. The main business entities in the IEF are account, branch, customer, transaction code and transaction type. Keys are allocated by a critical component, the *surrogate key server (SKServer)*.

Historically, IEF processed bank transactions in overnight batches. In a company's early implementation of the Surrogate Key Server (SKServer) (year 2000), the creation/lookup of surrogate keys was recognized as a major barrier to scalability. The implementation consisted of an *in-memory cache* of previously mapped entity-ID/surrogate-key pairs, which incurred a very high storage overhead, increasing both memory footprint and garbage collection activity. As the number of distinct business entities grew during the lifetime of the system, available memory required to process the overnight batch was exhausted eventually, causing the system to fail. We will refer to this version of the IEF as *Version 1*. In 2003, the problem was corrected in a new design of the SKServer that used a *file-based* surrogate key lookup for the assignment of keys for large sets of distinct business entities (such as accounts and customers), reducing the memory footprint. Over the subsequent few years, the number of distinct business entities continued to grow, requiring time-consuming I/O to process the batch of transactions. It then became apparent that the second design would eventually incur unacceptable processing times in the system. We will refer to this version of the IEF as *Version 2*. Therefore, in 2007, the Data Manager entered its third generation of design, in which transactions can also be processed in *real-time*, in addition to overnight batches. This last version of the IEF will be referred to as *Version 3*.

7.3 Case Study 1: A Prototyping Study

This case study, planned and executed during the first few months of the research, was designed to validate an early version of the framework and to provide quick feedback about the scalability analysis technique described in Chapter 4.

7.3.1 Objectives

At the time of the study, scalability was described as :

Concept: *"Scalability is a quality of software systems characterized by the causal impact that certain system characteristics have on certain measured system qualities, as these characteristics are varied over expected operational ranges. If the system can accommodate this variation in a way that is acceptable to the stakeholder, then it is a scalable system"*

The objective of the case study is to demonstrate that the scalability analysis technique provides the information required to characterize, analyze and compare the scalability of software systems, according to this definition.

The research questions formulated to achieve this objective are as follows:

RQ.1.1 Is it possible to measure characteristics of a software system quantitatively in such a way as to provide a characterization of its scalability according to the definition of scalability above?

RQ.1.2 Do the measurements support the systematic formulation of qualitative judgment about the scalability of the measured system, and the limitations of that scalability, that are consistent with the judgment an expert on the system would make?

RQ.1.3 Do the measurements support systematic comparison of the relative scalability of different system designs that are consistent with the comparative judgment an expert on the system would make?

7.3.2 Hypothesis

From the research questions, we formulated the following hypotheses:

H.1.1 The scalability analysis technique allows one to characterize quantitatively the scalability of a software system according to the definition above—that is, in terms of the impact that scaling characteristics of the application domain and the system design have on system qualities of interest.

H.1.2 The scalability analysis technique allows one to characterize systematically the satisfaction of the stakeholder with respect to system qualities when certain application domain and system characteristics vary over operational ranges. This characterization can be used to formulate a qualitative judgment about the system's scalability and its limits, and to compare the scalability of alternative system designs, in such a way that the result of the comparison is consistent with the judgment an expert on that system would make.

Finally, the case study relied on the following assumptions:

Assumption: The acceptability and utility of our definition of scalability has been convincingly argued through discussions in Chapters 1 to 3 and comparisons with other definitions in the computing literature.

Assumption: The stakeholder's judgment can be obtained consistently and without bias independently of the work performed in carrying out the case study.

7.3.3 Pilot Project

A pilot project should be representative of the type of project that would normally benefit from the evaluated technique. However, at the time of the study, we were aiming for a small-scale project that could give quick feedback about the scalability analysis technique. For this reason, we chose a retrospective study that simulates in a small scale, a real scalability problem faced by Searchspace: the replacement of original business entity identifiers with surrogate keys on the IEF. This replacement was performed, in the original system, by a critical component called the surrogate key server (SKServer). The project simulates Version 1 and Version 2 of the SKServer (implemented, respectively, in 2000 and 2003).

A retrospective study was chosen because it allows validation of the result of the analysis technique against what is known as fact about the system's scalability.

7.3.4 Planning

The case study was executed by the PhD candidate, with the cooperation of Searchspace staff. The PhD candidate played the role of the *analyst*. She was placed within the company and was given access to source code, documentation and the company employees. One staff member was allocated in weekly meetings to play the roles of the *stakeholder* and *expert*. The allocated staff member was directly involved with the architecture of both versions of the system that were being represented by the prototypes. Other staff members were also available for occasional consultation. More precisely, the analyst used the scalability analysis technique to characterize and compare the scalability of the prototypes of the Version 1 and Version 2 of the SKServer with respect to performance and resource usage metrics. Both prototypes offered identical functionality, but had different implementations. The case study involved the following steps:

1. Understanding of the actual problem faced by Searchspace in the period of 2000 to 2003;
2. Development of prototypes for both SKServer versions;
3. Setting up the required environment to run the study on the testing machine;
4. Instantiating the variables and functions for scalability analysis;
5. Executing both prototypes, collecting metrics that measure the system qualities of interest, and using this data to perform the scalability analysis as described in the technique;
6. Drawing conclusions about the study, comparing the analysis results to an expert's judgment.

7.3.5 Execution

We next describe the case study according to the execution plan above.

Step 1: Understanding the Problem

The analyst first familiarized herself with the problems faced by Searchspace during the period of time considered for the research. The expert described the original system, its architecture, the scalability problem it faced, and how the new version of the system solved this problem. The expert claimed that Version 1 of the SKServer was not scalable, as the growing number of distinct business entities eventually caused the system to fail. He also claimed that Version 2 of the SKServer was scalable, considering the load estimate at the time. This problem has been described in Section 7.2.

The stakeholder was mainly concerned with memory usage, which was dictated by the number of distinct business entities, rather than the number of transactions in a batch. This concern perhaps can be explained by the stakeholder's familiarity with the actual problem. Therefore, the stakeholder expressed his scalability concern as the system's ability to support a growing number of distinct business entities, while maintaining throughput and resource usage within acceptable boundaries.

Step 2: Creating the Prototypes

Both prototypes were built by the PhD candidate. For such, the code for Version 1 of the SKServer was isolated, removing functionality that allowed the tests to be performed independently of the rest of the IEF. The first prototype was therefore a stripped-down SKServer, which was left with a simple memory cache implemented as a Java Hashtable. Limitations on the JVM heap size (2GB in a 32-bit machine) restricted the number of entity-key maps that could be held in memory. In order to perform a comparative scalability analysis, we implemented another prototype for Version 2 of the SKServer that contained a cache that expanded to disk. This implementation limited the amount of heap that could be used to store entity-key maps. The cache implemented a least recently used (LRU) policy, where objects removed from the memory cache were kept as serialized objects on disk. Both implementations were discussed with the expert to ensure they approximated to the original systems.

Step 3: Setting up the Testing Environment

The prototypes were deployed in the machine described in Table 7.1.

Processor: Twin Intel(R) Xeon(TM)
CPU: 2.40GHz
Memory: 2GB
Disk: 4x 60GB - with RAID
Operating System: Red Hat Linux release 7.3 (Valhalla)
Database: Oracle version 9.2

Table 7.1: Case Study 1, Test machine specification.

Step 4: Instantiating the Framework

After conferring with the stakeholder, we determined that average throughput, maximum memory usage and maximum disk usage should be the dependent variables measuring performance and resource usage. Throughput and memory usage were primary concerns, as the system must be able to complete its analysis overnight and without running out of memory. Disk usage was also chosen as a dependent variable of interest because in adopting a file-based cache, the required space for holding the keys is shifted from memory to disk, and the stakeholder needed to analyze if this shift would impose a scalability problem.

We had learned in step 1 that the number of distinct business entities was the main factor affecting the qualities of interest, so they were chosen as a scaling independent variable. According to the stakeholder, the prototypes were required to handle up to 2 million distinct business entities. Since we were comparing two implementations of the same system whose difference was in the type of caching mechanism, this was chosen as a non-scaling independent variable. Available network bandwidth was also mentioned as a less important property that could affect the system throughput. However, because available network bandwidth did not directly relate to the mechanism for creating and looking up surrogate keys, it was assumed to be a nuisance variable.

Minimum throughput was targeted at 50 key substitutions per second. Memory and disk usage were limited by the JVM and available disk, respectively.[1] The lower and upper boundaries of the dependent variables are shown in Table 7.2.

	Lower boundary	**Upper boundary**
Average throughput	50 transactions/sec	—
Memory usage	—	2GB
Disk usage	—	73GB

Table 7.2: Case Study 1, Boundaries for dependent variables.

The variables instantiated for the analysis were as follows:

Questions:

Which SKServer prototype is more scalable with respect to average throughput, memory usage and disk usage, when the number of distinct business entities grows over time?

Independent variables:

Scaling variables:

Number of distinct entities (0 to 2 million)

Non-scaling variables:

Disk vs memory cache

Dependent variables:

Average throughput (at least 50 key substitutions/second)

Memory usage (up to 2GB)

Disk usage (up to 73GB))

The stakeholder has also declared that he would prefer the prototype that maximized throughput and minimized both memory and disk usage. The normalized preference functions for each one of these characteristics were:

Disk usage D(x): Disk usage was measured as the percentage of the total disk used.

$$\mathrm{D(x)} = \frac{\max_{disk}(\mathrm{V}_1,\mathrm{V}_2) - x}{\max_{disk}(\mathrm{V}_1,\mathrm{V}_2) - \min_{disk}(\mathrm{V}_1,\mathrm{V}_2)} \tag{7.1}$$

where $\max_{disk}(\mathrm{V}_1,\mathrm{V}_2)$ is the maximum disk usage computed throughout the execution of both prototypes, $\min_{disk}(\mathrm{V}_1,\mathrm{V}_2)$ is the minimum disk usage for both prototypes, and x is the percentage

[1]It was out of the scope of this study to apply a requirements engineering technique. Operational ranges of the system characteristics and the acceptable values of system qualities were tailored by the stakeholder according to the specification of the testing machine and assumed to be correct.

of total disk being used at the time the metrics were collected. As the preferred configuration minimizes disk usage, a smaller value represents a more desirable result. Hence, the function computes higher preference values to smaller disk usages.

Throughput T(y): Throughput was measured as the number of surrogate key substitutions per second. Since a throughput lower than 50 key substitutions per second is considered too low, the normalizing function penalizes the system by assigning a value of negative 10,000 in such situations.

$$\text{T(Y)} = \begin{cases} \text{- 10000}, & \text{if } y < 50 \\ \dfrac{\text{y - worst}_{perf}(\text{V}_1,\ \text{V}_2)}{\text{best}_{perf}(\text{V}_1,\ \text{V}_2)\text{ - worst}_{perf}(\text{V}_1,\ \text{V}_2)}, & \text{otherwise.} \end{cases} \tag{7.2}$$

where $\text{best}_{perf}(\text{V}_1,\ \text{V}_2)$ is the overall maximum throughput computed throughout the execution of both prototypes, $\text{worst}_{perf}(\text{V}_1,\ \text{V}_2)$ is the minimum throughput achieved in the execution and y is average throughput since the last collection of the metrics. The preferred prototype maximizes throughput, and therefore a higher value of both throughput and its representation as a preference indicate a more desirable result.

Heap size H(z): Heap size was measured as the total memory usage of the JVM.

$$\text{H(z)} = \frac{\text{max}_{mem}(\text{V}_1,\text{V}_2)\text{ - z}}{\text{max}_{mem}(\text{V}_1,\text{V}_2)\text{ - min}_{mem}(\text{V}_1,\text{V}_2)} \tag{7.3}$$

where $\text{max}_{mem}(\text{V}_1,\text{V}_2)$ is the overall maximum heap usage computed throughout the execution of both prototypes, $\text{min}_{mem}(\text{V}_1,\text{V}_2)$ is the overall minimum heap usage for both prototypes, and z is the memory usage measured as the Java heap footprint at the time the metrics were collected. The Java heap footprint is in fact less than the actual RAM footprint, as it only measures space consumed by Java objects. As with disk space, the preferred alternative minimizes heap size. Therefore, a smaller value represents a more desirable result, and the function computes higher preference values to smaller memory usages.

The stakeholder has declared that, at the time, he valued throughput and memory usage over disk usage, as the latter was becoming increasingly cheaper. The preferences for maximizing throughput and minimizing heap size were each expressed as being 10 times more important than minimizing disk usage. The (normalized) utility was calculated through Function 7.4, where n is the number of distinct business entities.

$$\text{U(n)} = \frac{\text{D(x) + 10 * T(y) + 10 * H(z)}}{21} \tag{7.4}$$

Step 5: Prototypes Execution

Tests consisted of continuously submitting transactions with distinct business identifiers to the surrogate key subsystem. Every new key incremented the counter for the number of distinct business entities. Each configuration was run 6 to 8 times and metrics for memory, disk and average throughput were collected periodically (every 5 minutes). Tests were run until the machine reached its physical limits or the system could no longer comply with its requirements.

Figures 7.1 and 7.2 show the metrics collected during the study. We plot the actual values rather than the preferences because it is easier to explain the behavior of the prototypes. Graphs on the left-hand side illustrate the metrics for the Version 1 prototype, while the right-hand side graphs represent the metrics for the Version 2 prototype.

Figure 7.1 plots the Java heap usage against the number of distinct business entities. Note that in the Version 1 prototype, the heap usage reaches almost 1.8GB. Since the Java heap footprint only measures space consumed by Java objects, the system was reaching the machine's actual memory limits of 2GB. Figure 7.2 plots throughput against the number of distinct business entities. In the Version 1 prototype, as the machine's memory usage nears 2GB, it starts to swap, bringing the throughput close to zero. Figure 7.3 plots disk usage against the number of distinct business entities. In this graph, one should look at the growth in the disk usage rather then the actual values (which simply reflect the amount of disk that were in use in the testing machine at the time tests were run). In the Version 1 prototype, surrogate key maps are kept in the memory cache, and hence there is not a significant growth in disk usage. The Version 2 prototype, on the other hand, uses a disk cache. This is reflected by the 10% growth in disk usage as the number of distinct business entities increases.

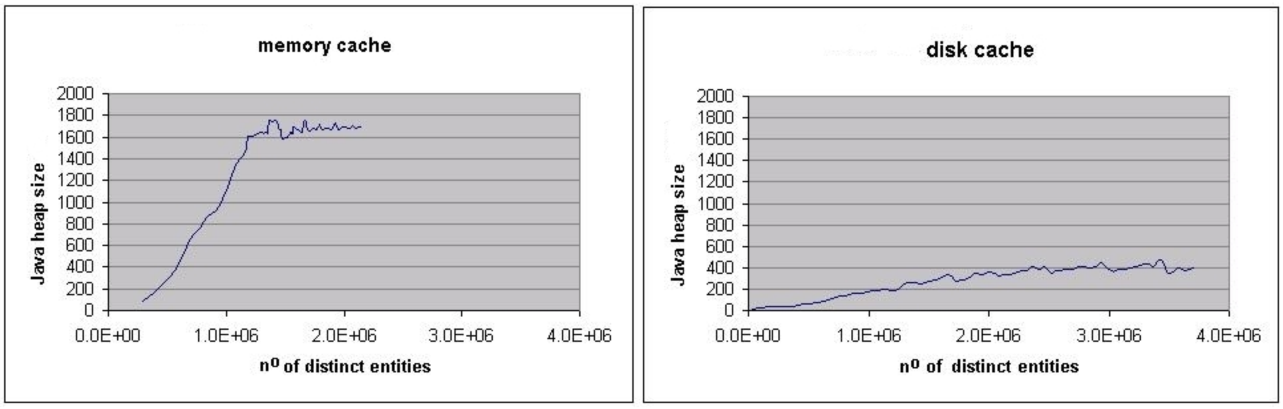


Figure 7.1: Case Study 1, Java heap vs number of distinct business entities.

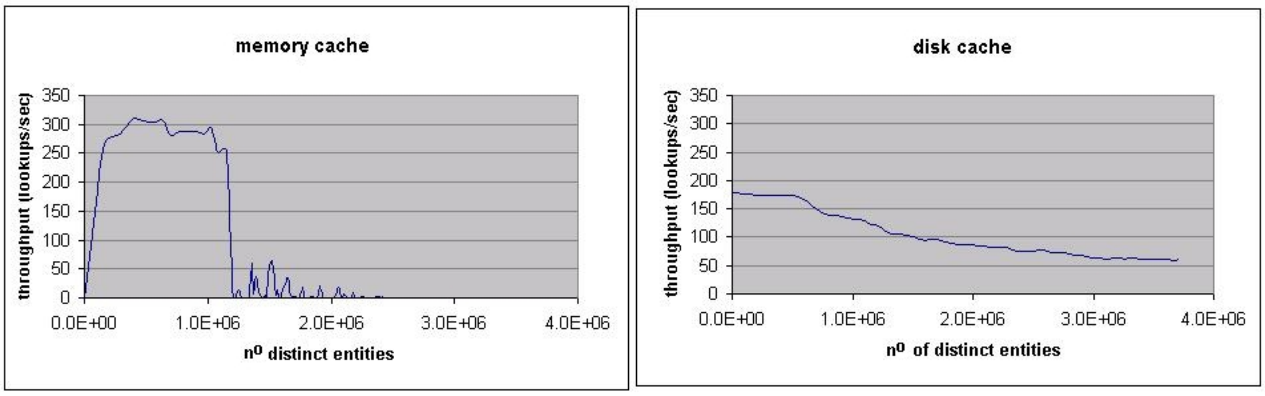


Figure 7.2: Case Study 1, throughput vs number of distinct business entities.

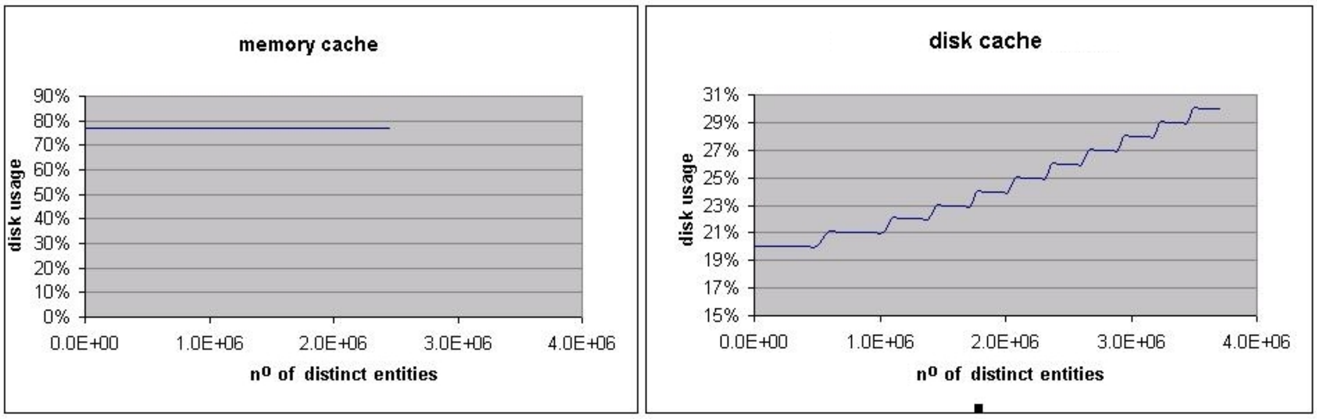


Figure 7.3: Case Study 1, disk usage vs number of distinct business entities.

Figure 7.4 plots the variation of the utility values for both prototypes against the measured range of distinct business entities. Note that as the number of distinct business entities increases over time, the utilities of both systems decrease. The utility of the Version 1 prototype starts very high, but it suffers a quick decrease, falling to zero when the number of distinct entities exceeds 1 million. This result is easily explained by the fact that in the first prototype, the system reaches the machine's physical memory limits, starting to swap, which brings the throughput below the accepted level. The occasional recovery shown in the graph (in the subsequent "impulses" between 1 and 2 million business entities) is explained by the disk swaps. The utility of the Version 2 prototype decreases smoothly, never reaching zero for the measured number of distinct business entities.

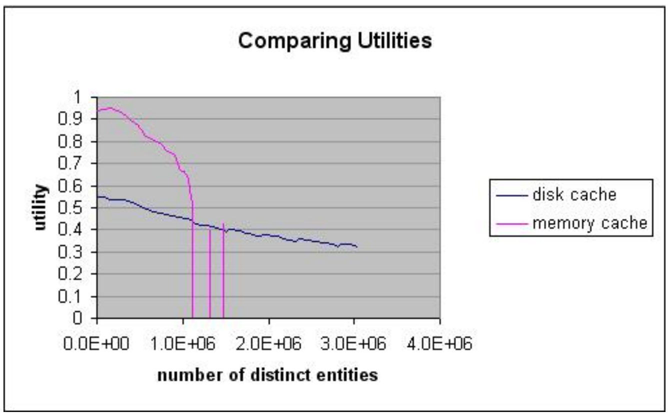


Figure 7.4: Case Study 1, utilities comparison.

Step 6: Drawing Conclusions

The hypotheses have been validated as follows:

Validation of Hypothesis H.1.1: The hypothesis states that the analysis technique should allow one to characterize quantitatively the scalability of a software system according to the definition of scalability. Validating this hypothesis requires evidence that the prototypes' scalability have been described through independent and dependent variables, and that the technique has produced quantitative data that shows the impact of variables representing scaling characteristics of the application domain and of the system design on variables representing the system qualities of interest.

The evidence provided by the case study is as follows:

- The instantiation of the scalability framework describing, in terms of independent and dependent variables, the scaling characteristics of the prototypes;
- Figures 7.1 to 7.4 demonstrating the causal impact of the scaling of the number of distinct business entities on the metrics representing system qualities of interest (memory usage, disk usage and average throughput).

These figures validate Hypothesis H.1.1 (and positively answer research question RQ.1.1), demonstrating that it is possible to measure the systems quantitatively in such a way as to provide a characterization of its scalability according to our definition of scalability.

Validation of Hypothesis H.1.2: The hypothesis states that analysis technique should allow one to characterize systematically the satisfaction of the stakeholder with the system qualities when certain application domain and system characteristics vary over operational ranges. Validating this hypothesis requires evidence that the stakeholder's scalability concerns have been described with preferences and utility functions, and data plots that provide solid confirmation of the expert's judgment have been produced.

The evidence provided by the case study is as follows:

- The instantiation of the scalability framework describing, in terms of preferences and utility functions, the stakeholder's scalability goals for the prototypes;
- Figure 7.4 demonstrating the variation in the utility of both prototypes, which confirm the expert's judgment expressed during the problem familiarization step.

According to the stakeholder's utility, the Version 1 prototype shows an unacceptable result when the system has to handle over 1 million distinct business entities. We can, therefore, conclude that the Version 1 prototype is not scalable with respect to throughput and memory usage for the measured ranges of distinct business entities—a result that is consistent with the expert's judgment. This evidence, therefore, provides a positive answer to research question RQ.1.2.

The graph also suggests that beyond the point of approximately 1 million distinct business entities, the Version 2 prototype presents a more desirable behavior than the memory-based prototype. From this observation, we conclude that, for the measured range of distinct business entities, the Version 2 prototype represents a more scalable solution with respect to the measured system qualities than the Version 1 prototype. This evidence therefore also positively answers research questions RQ.1.3, since the expert had judged the Version 2 solution more scalable than the Version 1 solution. By answering both questions, the study also confirms hypothesis H.1.2.

Finally, the qualitative judgment about the system scalability has been formed by following an number of steps, which included defining a scalability question, consulting with the stakeholder for the selection of variables and functions that represent their system qualities of interest and objectives, building and measuring the prototypes of the system, and using the collected data to instantiate the variables and functions, and to formulate an answer to the scalability question. The case study, therefore, also demonstrates the systematic nature of the framework

7.3.6 Evaluation

The critical evaluation of this study is divided into two parts: (1) threats to validity and (2) evaluation of the scalability analysis technique. Some of the threats discussed have been addressed in the second case study. The evaluation of the scalability analysis technique, which apply to both case studies, are discussed in Section 7.4.6.

Threats to Validity

This case study, carried out in the first few months of the research, had as an objective to provide quick feedback on the scalability framework at the time. For this reason, a number of simplifications were made, which imposed a few threats. Most importantly, the study used prototypes of the SKServer rather than the real system. Furthermore, a sufficiently powerful testing machine to support the load normally imposed on the IEF was not available, and the prototypes were run on a much less powerful UCL-owned machine deployed at Searchspace.

A threat to the *external/construct validity* of the study is that there is no guarantee that the prototypes used in the study mirrored the scalability problems faced by the real system. For this reason, the analysis results, which may well be a fair representation of the prototypes' scalability, may not be consistent with the results you might obtain for the real IEF. We tried to reduce this risk by stripping down the actual SKServer and discussing any new implementation with Searchspace staff to ensure that the prototypes were a fair representation of the IEF. Another threat to the *external validity* of the study is that we knew in advance that the number of business entities was the main scaling characteristic influencing the system's throughput, memory usage and disk usage. In the analysis of a newly built system, the choice of variables may not be so straightforward. In the scalability framework, this risk is mitigated by the requirements engineering techniques described in Chapter 5, which provide a systematic process for identifying obstacles to quality goals caused by the scaling of application domain characteristics.

The system's operational ranges and the acceptable values of system qualities were tailored by the stakeholder according to the specification of the testing machine, which also represents a threat to the *construct validity* of the study. This task was delegated to the stakeholders, who are experienced with running the IEF in less powerful machines and are, therefore, more able to estimate how the prototypes should perform in such an environment. Another threat to the *construct validity* was that upfront knowledge about the expert's judgment might bias the analyst to produce a prototype and generate data that would confirm what was already known. We tried to reduce this risk by consulting Searchspace staff when developing the prototypes and generating the synthetic data. Ideally, the staff member supervising the construction of the prototypes and the generation of data would have been different from the staff playing the roles of the stakeholder. However, unfortunately, no other staff with sufficient knowledge of the implementation of these old versions was available for participation on the study. These limitations have been addressed in the second case study.

A threat to the *internal validity* of the study is that we have not accounted for all characteristics that may affect the system qualities of interest. The only scaling characteristic considered in the analysis was the number of business entities, pointed out by the stakeholder as the main factor influencing the system's qualities of interest. We have also overlooked a few issues in this case study. The preference functions should have been normalized according to the requirements given by the stakeholder, not the minimum and maximum values assumed by the measured qualities. Although this does not invalidate the results, it might give a distorted impression of the stakeholder's satisfaction with the system. Finally, we could have artificially restricted the amount of memory and disk on the testing machine in order to better mimic the execution environment at the time of these IEF versions. These issues were also addressed in the second case study.

Despite the measures described above, there was still no guarantee that the study would be a fair representation of what happened with the real system. However, given that the case study was meant to provide quick feedback on the analysis technique in the early stages of the research, we felt that these were reasonable measures to take. Furthermore, a second case study was planned to address some of these threats.

7.3.7 Concluding Remarks

This section has described a case study using prototypes of a real-world system to provide quick feedback about the scalability analysis technique for characterizing, analyzing and comparing the scalability of software systems. The main points are:

- The case study aimed to validate the provide quick feedback about the analysis technique during the early stages of the research.
- The case study involved the PhD candidate and Searchspace staff.
- Using prototypes in the pilot project may diminish the contribution of the case study. Measures were taken to reduce this risk, when possible.
- The case study has demonstrated that it is possible to describe a software system's scalability according to the definition presented in the beginning of this section and to produce data plots that confirm an expert's judgment about the scalability of the system.

The described case study, by itself, is not sufficient to validate the analysis technique, as the pilot project uses prototypes of a typical IT system. Nevertheless, the study contributed to the research by validation of both hypotheses and by giving quick feedback on the analysis technique.

7.4 Case Study 2: A Comparison of Alternative Designs

Although that first study proves its hypotheses, there is no guarantee that the prototypes built for the study fairly reflected the scalability problems faced by Searchspace. A second case study was then designed to evaluate the previous research hypothesis, using the real IEF and more realistic set of testing data. In particular, the study compared Version 1 and Version 2 of the IEF, respectively implemented in 2000 and 2003. The former used a Java Hashtable object allocated in the Java Virtual Machine (JVM) heap as a cache. The latter used the Java Hashtable only for small sets of distinct entities, such as branch and transaction type, while large sets were stored in a disk cache. This study was reported on (Duboc et al., 2007).

7.4.1 Objectives

This study was designed to verify the results of the previous study. It, therefore, aimed to answer the same research questions as the previous study. These are:

RQ.2.1 Is it possible to measure characteristics of a software system quantitatively in such a way as to provide a characterization of its scalability according to the definition of scalability in Section 7.3.1?

RQ.2.2 Do the measurements support the systematic formulation of qualitative judgment about the scalability of the measured system, and the limitations of that scalability, that are consistent with the judgment an expert on the system would make?

RQ.2.3 Do the measurements support systematic comparison of the relative scalability of different system designs that are consistent with the comparative judgment an expert on the system would make?

7.4.2 Hypothesis

Likewise, this study used the same hypothesis as the previous one. These are:

H.2.1 The scalability analysis technique allows one to characterize the scalability of a software system quantitatively according to the definition in Section 7.3.1—that is, in terms of the impact that scaling characteristics of the application domain and system design have on system qualities of interest.

H.2.2 The scalability analysis technique allows one to characterize systematically the satisfaction of the stakeholder with respect to system qualities when certain application domain and system characteristics vary over operational ranges. This characterization can be used to formulate a qualitative judgment about the system's scalability and its limits, and to compare the scalability of alternative system designs, in such a way that the result of the comparison is consistent with the judgment an expert on that system would make.

Finally, the study relies on the same assumptions as previously:

Assumption: The acceptability and utility of our definition of scalability have been convincingly argued through discussions in Chapters 1 to 3 and comparison with other definitions in the computing literature.

Assumption: The stakeholder's judgment can be obtained consistently and without bias independently of the work performed in carrying out the case study.

7.4.3 Pilot Project

The project compares Version 1 and Version 2 of the IEF. Unlike the first study, which considered only the substitution of original business entities identifiers with surrogate keys, this study considered the four first stages of the IEF: validation, preprocessing, loading and migration.

A retrospective study was chosen because it allows one to validate the result of the analysis technique against what is known as fact about the system's scalability.

7.4.4 Planning

The case study was executed by the PhD candidate, with the cooperation of Searchspace staff. Both IEF versions offered identical functionality, but had different implementations. The case study involved the following steps:

1. Characterizing the application domain;
2. Deploying the two versions of the IEF on the testing machine;
3. Instantiating the variables and functions for the analysis;
4. Executing both IEF versions, collecting metrics that represent the system qualities of interest, and using the raw data collected to perform the scalability analysis as described in the technique;
5. Drawing conclusions about the study, comparing the results of analysis with the expert judgment.

7.4.5 Execution

We next describe the case study according to the execution plan devised above.

Step 1: Characterizing the Application Domain

The stakeholder has declared that banks using the IEF, at the time, had up to approximately 30 million accounts, 20 million customers, 1 thousand branches, 115 transaction codes and 130 transaction types (for a total of over 50 million business entities). The system was expected to process up to 30 million transactions in a time window of 8 hours, which corresponds to roughly one thousand transactions/second.

Synthetic data was generated by Searchspace staff following the same characteristics of a sample, taken from a large UK bank, with approximately 48 million bank transactions processed between 5^{th} January 2000 and 4^{th} March 2001. The sample included all the transactions performed by roughly one million randomly selected accounts, which corresponded to 3.2% of the total number of accounts in that particular bank. Although banks had in average 50% more accounts then customers, in the transactional batches the number of accounts were only 5% above the number of customers (possibly because many of the accounts are saving accounts, which are seldom involved in transactions). The remaining entities corresponded to only 0.003% of the total number of business entities in batches.

Step 2: Deploying Both Versions of the IEF

Version 1 and Version 2 of the IEF were deployed by the PhD candidate on the same machine used for the first case study (Table 7.1) and configured to run from the validation to the migration stages. Switching from one design to another was a matter of setting parameters in the system configuration files. In both designs, the size of the maximum JVM heap was set to 500MB to mimic the environment at the time these versions were in operation. The number of threads handling the data migration to the operational data store was configurable from 1 to 5 threads.

Step 3: Instantiating the Framework

The framework instantiation was almost identical to the first case study, with the exception of another scaling variable—the number of migration threads—which was considered by the stakeholder to be an important machine characteristic affecting throughput. Furthermore, the tests were run on a machine that was less powerful than the one normally used in the production environment.[2] Based on the experience of Searchspace's testing group, the maximum number of business entities was set to 5 million and the required throughput was set to at least 100 transactions/second, for a batch of 5 million transactions and considering that the system would run from the validation to the migration stages.

The instantiation of the framework was as following:

Questions:

Which IEF implementation is more scalable with respect to average throughput, memory usage and disk usage, when the number of distinct business entities grows over time?

Independent variables:

Scaling variables:

Number of distinct business entities (0 to 5 million)

Number of concurrent threads (1 to 5)

Non-scaling variables:

Memory vs disk cache

JVM memory size (500 MB)

Dependent variables:

Average throughput (at least 100 transactions/second)

Memory usage (0 to 500 MB)

Disk usage (0 to 24 GB)

The preferences stated by the stakeholder were still the same as in the previous case study, but they were modeled in such a way that the preference values were normalized against the expected bounds.

[2]The TCP-C benchmarks of the testing and production machines are 34,473 tpmC and 768,839 tpmC, respectively. Although we cannot make a direct comparison and state that the surrogate key server would run twenty times faster in the production environment, the benchmarks give us an indication of the difference in power of both machines.

As in the first case study, we took negative 10,000 to represent a penalty value. The preference functions were defined as follows:

Disk usage D(x): Disk usage was measured as the maximum percentage of the total disk used in the execution.

$$\text{D(x)} = \begin{cases} \text{- 10000}, & \text{if x} > 24 \\ \dfrac{\text{24 - x}}{\text{24 - 0}}, & \text{otherwise.} \end{cases} \tag{7.5}$$

where x is the maximum percentage of total disk used during the system execution (among the metrics collected periodically). As the preferred configuration minimizes disk usage, a smaller value of x represents a more desirable result.

Throughput T(y): Throughput was measured as the number of surrogate key substitutions per second.[3]

$$\text{T(y)} = \begin{cases} \text{- 10000}, & \text{if y} < 100 \\ \dfrac{\text{y - 100}}{\text{400 - 100}}, & \text{otherwise.} \end{cases} \tag{7.6}$$

where y is the average throughput during the system execution (calculated from the metrics collected periodically). The preferred prototype maximizes throughput, and therefore a higher value of y indicates a more desirable result.

Heap size H(z): Heap size was measured as the maximum memory usage of the system's JVM.

$$\text{H(z)} = \begin{cases} \text{- 10000}, & \text{if z} > 500 \\ \dfrac{\text{500 - z}}{\text{500 - 0}}, & \text{otherwise.} \end{cases} \tag{7.7}$$

where z is the maximum JVM memory usage during the system execution (among the metrics collected periodically). The preferred alternative minimizes heap size. Therefore, a smaller value of z represents a more desirable result.

Since the stakeholder maintained his opinion about the relative importance of system qualities, the utility function remained the same as in the first case study:

$$\text{U(n)} = \frac{\text{D(x) + 10 * T(y) + 10 * H(z)}}{21} \tag{7.8}$$

Step 4: Systems Execution

We have executed both systems, Version 1 (memory-based design) and Version 2 (disk-based design), varying the number of migration threads between one, three and five. Unlike the first case study, where the number of business entities increased throughout the system execution, in this study each version of the IEF was run to completion against the data batches of 5 million transactions containing an increasing number of distinct entities. The five batches containing, respectively, 1.5 thousand, 6 thousand, 50 thousand, 500 thousand and 50 million distinct business entities where generated using

[3]As no upper-limit was declared for throughput, 400 transactions/second (slightly above to the maximum observed throughput) was used as the upper bound.

a Searchspace in-house application, and following the characteristics in the sample described in Step 1. Each batch was run two or three times, individual measurements were recorded periodically (every 5 minutes), and the average number of the collected metrics over the executions for each batch was used. We next discuss and compare Version 1 and Version 2, both with 5 migration threads (as the systems obtained their best performance with this configuration).

Figure 7.5.a shows plots of JVM heap usage against the number of distinct business entities for Version 1 and Version 2. In Version 1, heap usage reaches 500MB, which was the maximum memory available to the JVM at the time. Therefore, the JVM runs out of memory at just a little over 4 million distinct business entities in memory, causing the system to fail. In Version 2, heap usage never exceeds around 20MB, as just the business entities with few distinct values are held in memory (roughly 1200 entities). Figure 7.5.b plots throughput against the number of business entities. In Version 1, as the JVM runs out of memory, the throughput goes to zero because of virtual memory swapping. In Version 2, throughput decreases only slightly, as the number of entities to be searched for a lookup increases. Disk usage, although a stakeholder concern, was 1%–2% of the total available space in both versions, proving not to be a problem.

Figure 7.6 shows a plot of the utility against the measured range of distinct business entities for Version 1 and Version 2. The plot shows that the utility of Version 1 exceeds that of Version 2 for most of the measured range, but drops to zero after a critical point of approximately 4 million distinct business entities. The file-based implementation has its utility dropping only slightly and very smoothly from roughly 0.7 to 0.6 as the number of distinct business entities increase. Although the study cannot prove that the disk-based design scales to the required production data load, it is sufficient to demonstrate the different scalability characteristics of the two designs.

Step 5: Drawing Conclusions

The hypotheses have been validated as follows:

Validation of Hypothesis H.2.1: The hypothesis states that the analysis technique should allow one to characterize quantitatively the scalability of a software system according to the definition of scalability. Validating this hypothesis requires evidence that the systems' scalability have been described through independent and dependent variables, and that the technique has produced quantitative data that shows the impact of variables representing scaling characteristics of the application domain and of the system design on variables representing the system qualities of interest. The resulting data should be consistent with known facts about the system.

The evidence provided by the case study is as follows:

- The instantiation of the scalability framework describing, in terms of independent and dependent variables, the scaling characteristics of both versions of the system;
- Figures 7.5 and 7.6 demonstrating the causal impact of the scaling of the number of distinct business entities on the metrics representing system qualities of interest (memory usage, disk usage and average throughput).

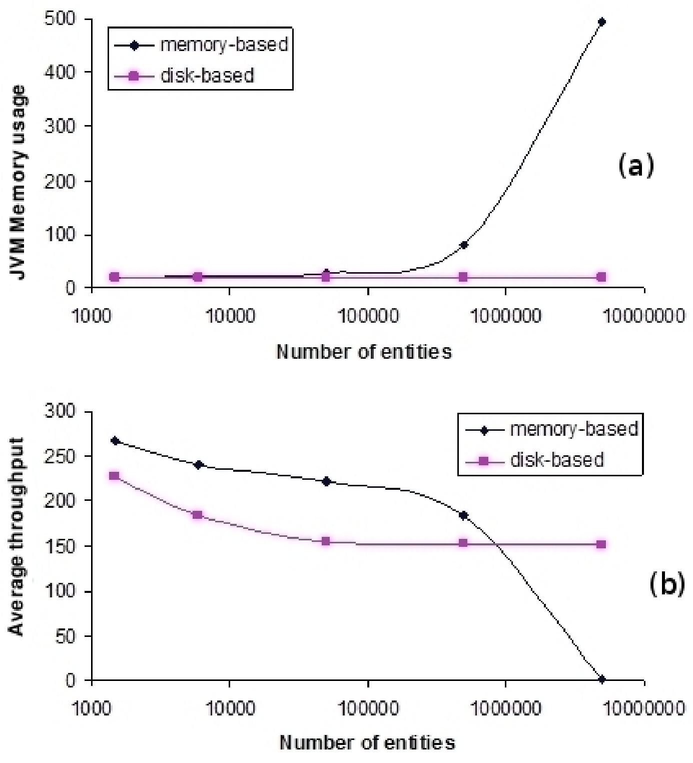


Figure 7.5: Case Study 2, measured system qualities.

Similarly to case study 1, the graphs in these figures confirm Hypothesis H.2.1 (and positively answer research question RQ.2.1), demonstrating that it is possible to measure characteristics of both versions of the IEF quantitatively in such a way as to provide a characterization of its scalability according to our definition of scalability.

Validation of Hypothesis H.2.2: The hypothesis states that the analysis technique should allow one to characterize systematically the satisfaction of the stakeholder with the system qualities when certain application domain and system characteristics vary over operational ranges. Validating this hypothesis requires evidence that the stakeholder's scalability concerns have been described with preferences and utility functions, and data plots that provide solid confirmation of the expert's judgment have been produced.

The evidence provided by the case study is as follows:

- The instantiation of the scalability framework describing, in terms of preferences and utility functions, the stakeholder's scalability goals for both versions of the system;
- Figure 7.6 demonstrating the variation in the utility of both systems, which confirm the expert's judgment expressed during the problem familiarization step.

Given that the utility function was modeled in such a way that it would receive a negative value if the system requirements were not met, we can conclude that Version 1 is not scalable with respect to throughput and memory usage for the measured range of business entities, which is consistent

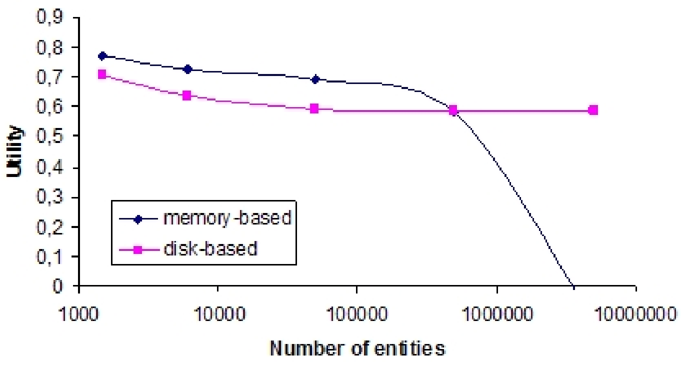


Figure 7.6: Case Study 2, utility comparison.

with the stakeholder judgment on that version. This result provides a positive answer to research question RQ.2.2.

In addition, no extrapolation was necessary to show that Version 2 was more scalable than Version 1. As the system has to handle on average around 36.66 million entities, and as the JVM memory at the time was limited to 500MB, we can conclude that Version 1 should have been withdrawn from consideration. We also know as fact that Version 2 can handle the required batches with 30 million transactions and 50 million distinct entities. This result answers research question RQ.2.3 positively. By answering both questions RQ.2.2 and RQ.2.3 positively, the study also confirms hypothesis H.2.2.

As in the previous study, qualitative judgment about the system scalability has been formed by following number of steps for the definition of the scalability question, the selection of variables and functions that represent the system qualities of interest, the collection of data, and the instantiation of the variables and functions to provide a scalability answer. This second case study, therefore, also shows the systematic nature of the framework.

7.4.6 Evaluation

The critical evaluation is divided in two parts: (1) threats to validity and (2) evaluation of the analysis technique.

Threats to Validity

In hindsight, Version 2 may appear to be obviously superior to Version 1. One may speculate that Searchspace's problem was in correctly estimating the load and that the analysis technique would bring little additional benefit, which may represent a threat to the *external validity* of the study. In fact, simple back-of-the-envelope calculations could have warned developers about the problem. Surrogate keys for business entities were held in objects of 16 bytes and were intended to be stored in a Java HashMap. A HashMap has roughly 116 bytes with an overhead of 24 bytes per object. Even if the system did nothing but hold 50 million of these objects in memory, it would require 1.5GB of heap space, which was more than the 500MB available. At the time, however, the problem was not obvious at all. Furthermore, the study intended to show that it is possible to characterize the scalability of a real-world software system quantitatively and the satisfaction of the stakeholder with respect to system qualities when these characteristics are varied over operational ranges. Whether the analyst uses prototyping, modeling or back-of-the-envelope calculations to quantify these variables is irrelevant for the technique. Furthermore, the scalability framework advocates a proactive approach to scalability, encouraging the analyst to consider scalability from the early stages of software development. We can therefore reasonably speculate that such a framework would have warned developers of the scalability problem faced by the IEF.

A problem we tried to tackle in the second study was to have a more realistic data set. However, customer data could not be viewed by the PhD candidate. In order to address this problem, synthetic data, generated based on real characteristics, was provided by Searchspace staff. The sample used to generate the data was small (containing only 3.2% of the accounts for that particular bank), and the ratio between the business entities in that sample was assumed to hold for different batches of transactions. The synthetic data produced was assumed to be a fair representation of Searchspace's actual customer data, although in reality this cannot be guaranteed, representing a threat to the *construct validity* of the system.

Despite addressing some of the limitations of case study 1, this case study shares a few of its threats. We still did not have a powerful enough machine to run the tests and, as in the first case study, the operational ranges of the system characteristics and the acceptable values of system qualities were tailored by the stakeholder according to the specification of the testing machine. In order to reduce the risk to the *construct validity* of the study, we counted on the experience of Searchspace's testing group for this task, who routinely performed tests on less powerful machines. Yet, the size of each individual transaction (and of business entities) was maintained, and the memory in the testing machine was limited to 500MB to mimic the production environment at the time. Therefore, the memory exhaustion seen in the study does reflect what, in reality, has happened with the IEF. In addition, as in the first case study, the analysis was performed by the PhD candidate, who may be biased to produce a positive result. Therefore, despite using a typical IT system that would normally benefit from the technique, the case study does not use a typical analyst, which is also a threat to the *construct validity* of the study. The risk of a positively biased result was reduced by constantly consulting Searchspace staff about the execution of the case study. Unfortunately, no other person was available to play the role of the analyst. Finally, this study also does not account for all scaling characteristics that may affect the system qualities of interest. This threat to the *internal validity* of the study is mitigated in the scalability framework by our techniques for elaborating scalability requirements.

Evaluation of the Analysis Technique

Case studies 1 and 2 highlighted some limitations of the proposed technique in a real-world setting:

Choosing variables and functions: Being a retrospective study, the analysis was simplified by the fact that the scalability problem was well-known. Such familiarity has probably helped the stakeholder to choose the relevant scaling variables and system characteristics—a task that would not have been so simple for a new system. Choosing appropriate variables and functions is crucial to the analysis. This observation motivated us to develop the requirements engineering technique described in Chapter 5.

Early scalability analysis: Similarly, the fact that both designs were already implemented allowed reliable metrics to be collected and analyzed. There will be occasions when an analysis of a yet unimplemented system will be required. The importance of quality predictions in early stages of the development lifecycle is advocated in software engineering (Smith and Williams, 2001). Although further research is needed to confirm this, at least theoretically, raw data could be produced for the analysis through models, such as queuing networks and Petri nets.

Design of scalability tests: We knew as fact that the growth in the number of business entities (rather than another scaling characteristic) had been responsible for the scalability problem of Version 1 of the IEF. This previous knowledge made testing straightforward. However, when analyzing a new system, there may be multiple scaling characteristics affecting a number of system qualities. The technique is still lacking support for systematically deriving a set of tests cases for the analysis.

Extrapolation beyond measured ranges: The case studies have shown that Version 2 was more scalable than Version 1, but they have not proved that it was scalable with respect to performance. Doing so requires either a sufficiently powerful infrastructure to run the system against the full range of distinct business entities or extrapolation of the analysis results. Extrapolation is subject to substantial uncertainty and is very challenging when the utility combines variables with distinct scaling characteristics. For the case study, extrapolation was not necessary to demonstrate the superiority of the Version 2's design. Realistically, extrapolation often will be required, and is left as a topic of future research.

Despite the limitations, the case studies have been successful in demonstrating important benefits of the technique in a real-world setting:

Quantitative characterization of system scalability and stakeholder preferences: The case studies have shown that the technique can quantitatively characterize a software system's scalability according to the definition in Section 7.1 and the satisfaction of the stakeholder with respect to system qualities when system characteristics vary over operational ranges.

Clear and objective comparison of the scalability of alternative system designs: The case studies have also demonstrated the suitability of the technique for clearly and objectively comparing the scalability of alternative designs in a large real-world system.

The quantitative characterization produced by the technique allows one to describe the scalability of a software system clearly and objectively. As a result, the technique tackles the problem of intuitions, ambiguities and inconsistencies underlying the notion of scalability in computing. Finally, by forcing scalability concerns to be clearly expressed as variables and functions, the technique encourages software developers to think carefully about the subject, possibly bringing into evidence problems that might otherwise be overlooked.

7.4.7 Concluding Remarks

This section has described a case study to validate the scalability analysis technique in a real world setting. The main points are:

- The case study demonstrated that it is possible to describe a software system's scalability according to the definition presented in Section 7.3.1, and produced data plots that confirm what is known as fact about the real system's scalability.
- The case study involved the PhD candidate and Searchspace staff. Experienced Searchspace staff, not involved in the first study, has helped the generation of synthetic data.
- The case study validated the stated hypotheses for a real-world system. The development of a requirements engineering technique was a direct result of the limitations observed in this study.

7.5 Case Study 3: Elaborating the Scalability Requirements of the IEF

A third case study was planned to validate the technique described in Chapter 5. Part of this study was reported on (Duboc et al., 2008).

7.5.1 Objectives

The objective of this case study was to validate the requirements engineering technique described in Chapter 5 against a real-world system, and to demonstrate the derivation of variables and functions for a scalability analysis from the system's scalability requirements. The research questions formulated to achieve these objectives are as follows:

RQ.3.1: How to describe, model, and analyze scalability requirements of software systems systematically?

RQ.3.2: Is it possible to derive variables and functions for scalability analysis systematically from a system's scalability requirements?

7.5.2 Hypothesis

In order to answer the research question RQ.3.1, we have applied the technique described in Chapter 5 to elaborate the requirements of a real-world system. Systematically describing, modeling and analyzing scalability requirements demands form the technique the properties stated in the following hypothesis:

H.3.1: The technique is applicable to large-scale, industrial projects.

H.3.2: The technique allows one to identify systematically important scaling and non-scaling characteristics of the application domain that may be missed if no systematic technique is applied.

H.3.3: The technique allows one to identify systematically a set of scalability risks, some of which may be missed if no systematic technique is applied, and to link each scalability risk to the system goals it may impact.

H.3.4: The technique allows one to explore systematically a wide range of alternative ways to deal with scalability risks, and envision solutions that may be missed if no systematic technique is applied.

Question RQ.3.2 is addressed by the step 6.1.5 of the method described in Chapter 6. The study hypothesis formulated to answer this question is as follows:

H.3.5: The method allows one to derive systematically from a goal model a set of variables and functions for a scalability analysis.

Our requirements engineering activities took place in parallel with the third major re-design of the IEF and, for this reason, the requirements for this new version, arguably, had been defined (albeit not in a goal-oriented form). Therefore, a limited comparison of the resulting goal model is made against the original requirements specification document for the system and against a goal model developed using the standard KAOS techniques.

Therefore, the case study relied on the following assumptions:

Assumption: The original requirements specification document for the system is unbiased and considered to be complete and comprehensive by its authors.

7.5.3 Pilot Project

As we started to consider KAOS as a technique for elaborating scalability requirements, we constructed goal models for small portions of the IEF, which were circulated within Searchspace. A few months later, the IEF entered a major re-design (for Version 3) and the project manager responsible for this new IEF version suggested that we applied KAOS in parallel with the development activities. The IEF was then chosen as the pilot project for evaluating the suitability of our extension to KAOS for elaborating scalability requirements.

The IEF is a complex, real-world system, and is therefore representative of other systems that would benefit from the proposed technique. Elaborating the requirements of the new IEF version included all the complexities normally present on such activity: a wide variety of goals, multiple input sources, multiple stakeholders' views, inconsistent and conflicting goals, alternative solutions, a complex application domain, and changing goals.

7.5.4 Planning

The case study was executed by the PhD candidate, with the cooperation of Searchspace staff. The PhD candidate played the role of the *requirements analyst*. She had worked for three a half years at Searchspace and therefore had some domain knowledge. She had no previous experience with KAOS. KAOS was learned through technical papers and discussions of the evolving model with an expert researcher in the area. The role of the *stakeholders* was played mainly by Searchspace's global director and by the IEF's architecture team leader, with the occasional input of other Searchspace staff who were available for questions.

The case study involved a number intertwining steps:

Step 1: Familiarization with the goals for the new version of the IEF and construction of the standard KAOS goal model;

Step 2: Model refinement through the systematic application of scalability goal-obstacle analysis;

Step 3: Instantiation of variables and functions required for scalability analysis from selected goals;

Step 4: Drawing conclusions about the study, including a comparison with the IEF's original requirements specification document and standard goal model.

7.5.5 Execution

This section describes the case study according to the execution plan devised above. The final model contains 145 goals and 11 agents. Due to the scale of the goal model and the sensitivity of the information it contains, it cannot be fully described in this thesis. However, some of its goals, assumptions, obstacles and resolutions have been discussed in Chapters 5 and 6, and will be described further next. The quantitative data presented in the goals throughout this chapter have been changed in order to protect the company's proprietary information.

Step 1: Familiarization with the goals of the new version of the IEF and construction of the standard goal model

The elaboration of scalability requirements combined a number of activities, such as interviews, brainstorming, data characterization, and document review. Documents included the product specification of the previous IEF version, the business requirements specification and use cases documents originally developed by Searchspace for this new version of the system, and marketing documents describing the main characteristics of the company's customer base.

The business requirements specification document contains 44 requirements divided into architecture, transaction processing, data handling, user interface, rules testing, development timetable, backwards-compatibility and performance. Requirements were described in natural language, usually in one or two sentences. For example, the requirement on the processing of transactions has been stated as "Rules can be applied real-time as the records arrive or as part of a batch process as defined by the rule".

Nearly all requirements in the document were functional. The document contained exactly two requirements under the "performance" category. One stating that the new IEF version should run in a particular platform, and the other quantifying the expected number of transactions to be processed per day—a number that had not been justified in the document. The latter requirement has been stated as "It should be capable of processing 200,000 transactions a day."[4]

[4]The actual number of transactions have been modified to protect Searchspace's intellectual property.

As originally stated, the requirements of the IEF were more a statement of idealized goals rather than precise requirements with clearly understood objectives.

We next describe a portion of the IEF goal model developed through the standard application of KAOS.

Standard KAOS goal model:

We started the construction of the standard KAOS goal model by looking at the interests of Searchspace's customers and the nature of their operating environments that most affected the IEF. At a very high level, a bank has strategic goals such as Maintain[Bank Reputation], Avoid[Inconvenience to Account Holders] and Avoid [Loss Due to Fraudulent Transactions]. All these goals receive a positive contribution from the goal Achieve[Fraudulent Transactions Detected and Acted Upon], as shown in Figure 7.7.

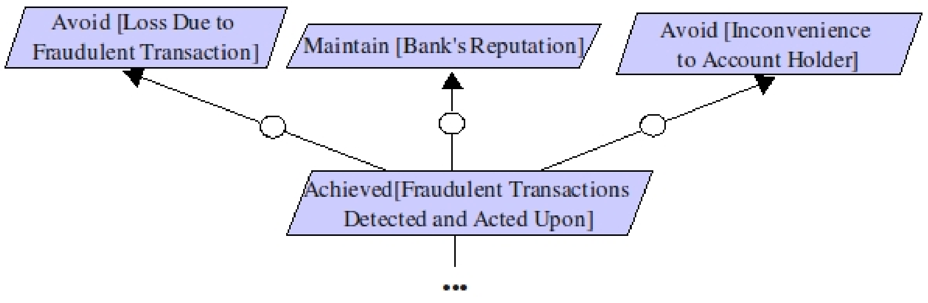

Avoid [Loss Due to Fraudulent Transaction]
Maintain [Bank's Reputation]
Avoid [Inconvenience to Account Holder]
Achieved[Fraudulent Transactions Detected and Acted Upon]
...

Figure 7.7: Case Study 3, High-level, strategic IEF goals.

Detecting and acting upon fraudulent transactions require that suspicious transactions are identified, investigated, and resolved when a fraud is confirmed. Resolving a fraudulent transaction includes canceling it before completion and reversing it after completion. The responsibility for resolving fraudulent transactions is assigned to the agent Bank Staff. Furthermore, financial services regulation requires information about fraudulent transactions to be recorded. The goal Achieve[Fraudulent Transaction Detected and Acted Upon] is therefore AND-refined into the goals Achieve[Possible Fraudulent Transactions Alerted Quickly], Achieve[Alerts Stored] and Achieve[Alerts Investigated and Acted Upon Quickly], as illustrated in Figure 7.8. Alerts are stored in an external data storage, represented in the goal model by the agent Data Storage.

The IEF recognizes fraudulent behavior both by comparing transactions with known fraudulent patterns and by learning what is unusual behavior compared with other transactions of the same kind. For simplicity, we consider only the first case. In order to satisfy the goal Achieve[Alerts Investigated and Acted Upon Quickly], this goals is refined into an assumption Fraudulent Transactions Follow Known Patterns, and the goals Achieve[Fraudulent Patterns Recorded] and Achieve[Transactions Matching Known Fraudulent Patterns Alerted]. Recording known fraudulent patterns is accomplished by encoding, testing and storing patterns into the system, which is represented in the model by an AND-refinement of the goals Achieve[Known Fraudulent Patterns Encoded], Achieve[Fraudulent Patterns Tested Against Real Transactions] and Achieve[Fraudulent Patterns Stored and Activated]. Finally, Searchspace stores incoming transactions that are then used to test new fraudulent patterns. This is represented in the model by an AND-refinement of the

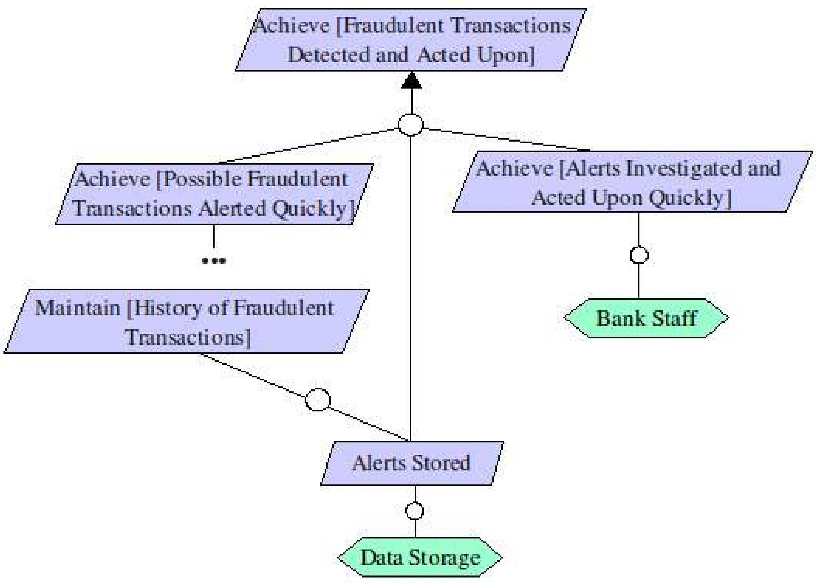

Achieve [Fraudulent Transactions Detected and Acted Upon]
Achieve [Possible Fraudulent Transactions Alerted Quickly]
Achieve [Alerts Investigated and Acted Upon Quickly]
...
Maintain [History of Fraudulent Transactions]
Bank Staff
Alerts Stored
Data Storage

Figure 7.8: Case Study 3, Fraudulent transactions detected and acted upon.

goal Achieve[Fraudulent Patterns Tested Against Real Transactions] into the goals Achieve[Incoming Transactions Stored] and Achieve[Fraudulent Patterns Tested Against Stored Transactions]. Fraudulent patterns are encoded into the system by Searchspace Staff. Incoming transactions and fraudulent patters are stored in the Data Storage and then tested using the Alert Generator. All these goals are shown in Figure 7.9.

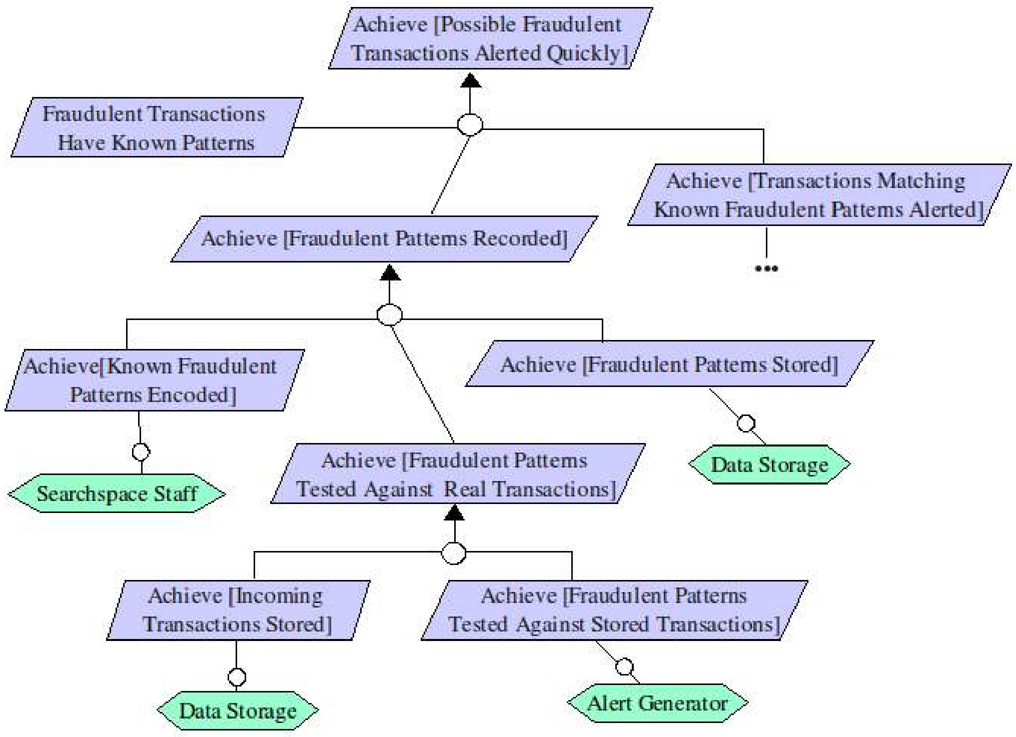

Achieve [Possible Fraudulent Transactions Alerted Quickly]
Fraudulent Transactions Have Known Patterns
Achieve [Transactions Matching Known Fraudulent Patterns Alerted]
Achieve [Fraudulent Patterns Recorded]
...
Achieve[Known Fraudulent Patterns Encoded]
Achieve [Fraudulent Patterns Stored]
Searchspace Staff
Achieve [Fraudulent Patterns Tested Against Real Transactions]
Data Storage
Achieve [Incoming Transactions Stored]
Achieve [Fraudulent Patterns Tested Against Stored Transactions]
Data Storage
Alert Generator

Figure 7.9: Case Study 3, Possible fraudulent transaction alerted.

There are two main approaches for alerting transactions that match known fraudulent patterns:

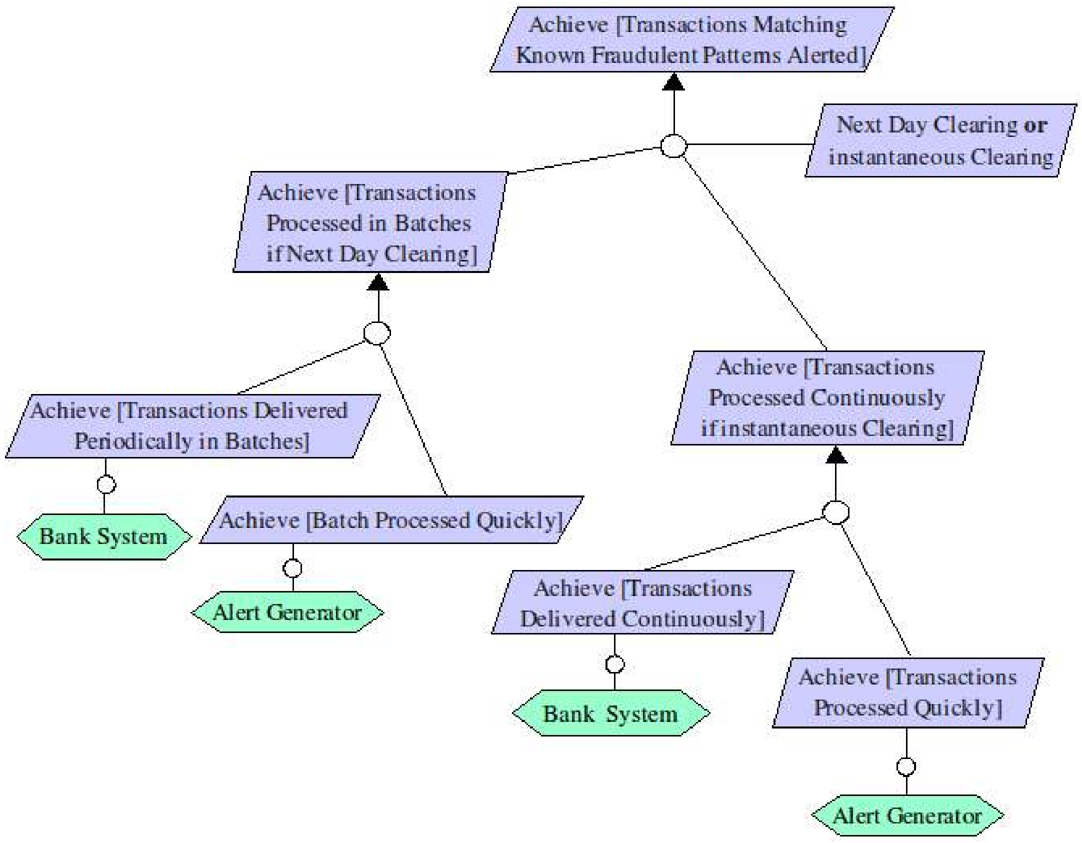

Achieve [Transactions Matching Known Fraudulent Patterns Alerted]
Next Day Clearing or instantaneous Clearing
Achieve [Transactions Processed in Batches if Next Day Clearing]
Achieve [Transactions Processed Continuously if instantaneous Clearing]
Achieve [Transactions Delivered Periodically in Batches]
Bank System
Achieve [Batch Processed Quickly]
Alert Generator
Achieve [Transactions Delivered Continuously]
Bank System
Achieve [Transactions Processed Quickly]
Alert Generator

Figure 7.10: Case Study 3, Transactions matching known fraudulent patterns alerted.

continuously and overnight. In the former, transactions are streamed by the bank system into the IEF and processed as soon as they arrive. In the latter, transactions are provided periodically and processed in data batches. The selection between them depends on the type of fraud being addressed and the upstream banking processes. For example, the clearing process for some types of transactions (e.g., cheque transactions) is normally performed in the next working day, so they can be processed overnight. Other transactions (e.g., debit card online transactions) may require instantaneous clearing. The goal Achieve[Transactions Matching Known Fraudulent Patterns Alerted] is therefore refined using the pattern *decomposition-by-cases* (van Lamsweerde, 2008), as in Figure 7.10. Note that transactions either require next day or instantaneous clearing, never both. Therefore, satisfying the goal Achieve[Transactions Processed in Batches if Next Day Clearing] or the goal Achieve[Transactions Processed Continuously if Instantaneous Clearing] satisfies the parent goal. The former goal is AND-refined into the expectation Achieve[Transactions Delivered Continuously] and the goal Achieve[Transactions Processed Quickly]. The latter goal is AND-refined into the expectation Achieve[Transactions Delivered Periodically in Batches] and the goal Achieve[Batch Processed Quickly]. Transactions are delivered by the agent Bank System, and processed by the agent Alert Generator.

There are many other goals in the KAOS goal model for the IEF, such as Achieve[Learn Usual Customer Behavior], Achieve[Backward Compatibility with Previous Versions], Achieve[Integration with Other Searchspace Systems], Minimize[Total Cost of Ownership],and Achieve[Deployment of Multiple Products on the Same Hardware]. Nevertheless, for the purpose of this thesis, the description of the IEF model stops here. The next section demonstrates the identification and resolution of scalability obstacles associated with the goals described above.

Step 2: Model refinement through scalability goal-obstacle analysis

Starting from the standard KAOS goal model produced in the previous step, the scalability goal-obstacle analysis consists of (1) systematically *identifying obstacles* that may obstruct the satisfaction of the goals, requirements, and expectations elaborated so far; (2) *assessing* the likelihood and criticality of those obstacles; and (3) *resolving* them by modifying existing goals, requirements, and expectations, or generating new ones so as to prevent, reduce or mitigate the identified obstacles. These steps are illustrated below:

Identifying Obstacles

This step requires systematically checking for scalability obstacles to all the requirements and expectations in the model. That is done by comparing, for each agent, the *capacity* of this agent against the *load* imposed on it by its assigned goals. We next exemplify the identification of such obstacles for the following three agents: Alert Generator, Data Storage and Bank Staff. A list of scalability obstacles in the goal model described in the previous section is given in Table 7.3.

Scalability obstacles to goals under the responsibility of the Alert Generator

The Alert Generator is responsible for three of the leaf goals presented above: Achieve[Batch Processed Quickly], Achieve[Transactions Processed Quickly] and Achieve[Fraudulent Patterns Tested Against Stored Transactions], as shown by the agent load diagram in Figure 7.11.

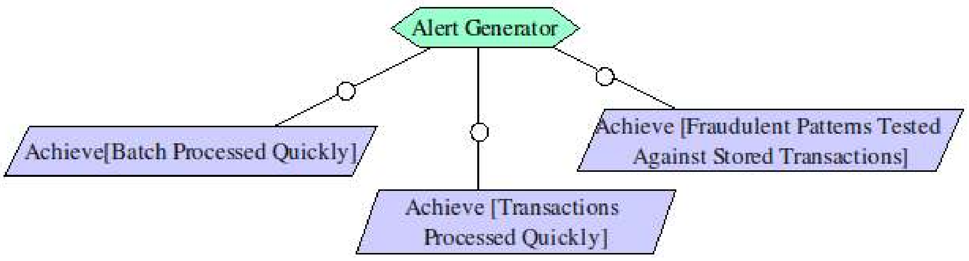

Alert Generator
Achieve[Batch Processed Quickly]
Achieve [Transactions Processed Quickly]
Achieve [Fraudulent Patterns Tested Against Stored Transactions]

Figure 7.11: Case Study 3, Alert Generator agent responsibilities.

In all these goals, each transaction is compared against a set of known fraudulent patterns. We refer to this process as "txn check". We define the Alert Generator *capacity* in terms of the average number of txn checks per second it can support. The *load* imposed on this agent (also defined in terms of the average number of txn checks per second) depends on many application domain characteristics, such as the number of transactions, the number of fraudulent patterns, the type of fraudulent patterns, and so on. We next look at the specification of each of these goals to determine this load:

Goal 7.1

Goal Achieve [Batch Processed Quickly]

Category Performance

Definition Batches of bank transactions provided by the bank source system should normally be processed in less than 8 hours. Processing consists of generating alerts on the transactions in the batch that match known fraudulent patterns stored in the system.

Quality Variable: processingTime: Time

Definition The time required from reading the transactions to generating alerts on all transactions that match one or more fraudulent patterns.

Sample Space Set of batches of bank transactions submitted by the bank source system.

Objective Functions At least 9 out of 10 batches should be processed within 8 hours, and all batches should be processed within 9.6 hours.

Name	Definition	Mode	Must
% of Batches Processed in 8 hours	Pr(processingTime $\leq$ 8h)	Max	90%
% of Batches Processed in 9.6 hours	Pr(processingTime $\leq$ 9.6h)	Max	100%

Responsibility Alert Generator

Goal 7.2

Goal Achieve [Transactions Processed Quickly]

Category Performance

Definition Each bank transaction provided by the bank source system should be processed immediately and in less than 0.5 milliseconds. Processing consists of generating an alert on any transaction that matches one or more known fraudulent patterns stored in the system.

Quality Variable processingTime: Time

Definition The time required from reading the transaction to generating an alert if it matches a fraudulent pattern.

Sample Space: Set of transactions submitted by the bank source system.

Objective Functions At least 8 out of 10 transactions should be processed within 0.5 milliseconds each, and every transaction should be processed within 2 milliseconds.

Name	Definition	Mode	Must
% of Txns Processed in less than 0.5 ms	Pr(processingTime $\leq$ 0.5ms)	Max	80%
% of Txns Processed in less than 2 ms	Pr(processingTime $\leq$ 2ms)	Max	100%

Responsibility Alert Generator

Goal 7.3

Goal Achieve [Fraudulent Patterns Tested Against Stored Transactions]

Definition New patterns of fraudulent transactions encoded by Searchspace staff should be tested against real transactions stored in the system. The test should certify that the new fraudulent patterns have been correctly encoded; that is, that alerts are generated for the transactions in the test data expected to match the new pattern and that are not generated for transactions that are not expected to match the pattern.

Responsibility Alert Generator

These goals, as stated, do not identify the ranges of application domain characteristics that will determine their load. The characteristics of the application domain whose ranges have been overlooked in the original model are the: number of transactions in a batch, number of distinct business entities in a batch, number of transactions per second, number of distinct business entities per second, number of transactions in test data, number of distinct business entities in test data, number of fraudulent patterns (to be compared against incoming transactions), and number of new fraudulent patterns in any given day (to be tested against stored transactions). The type of the fraudulent patterns had also been overlooked. Some fraudulent patterns, such as the ones which consider historical information about the account or the behavior of peer accounts take longer to process than others that are simple comparisons against a blacklist of accounts, countries or institutions. [5] The type of fraudulent pattern is represented in the goal model as a standard KAOS domain assumption named Types of Fraudulent Patterns.

Therefore, the load on each of these goals can be considered infinite, inducing the scalability obstacle Number of Txn Checks Exceeds Alert Generator Processing Speed. This obstacle is illustrated in Figure 7.12.

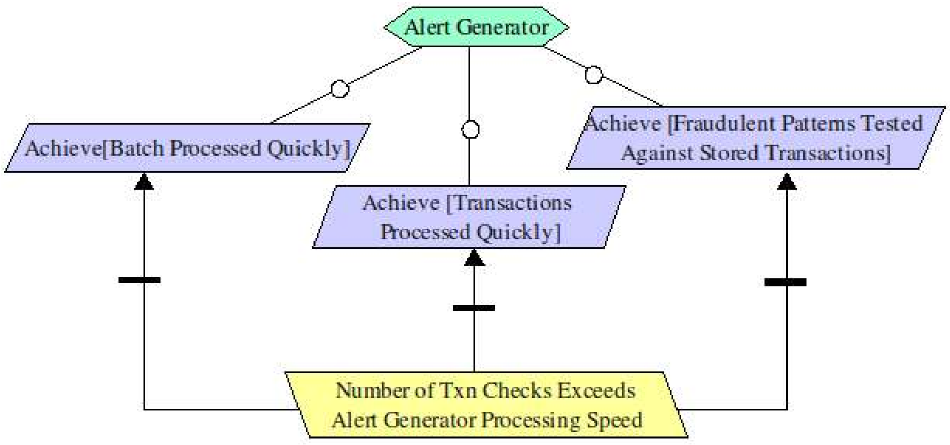

Alert Generator
Achieve[Batch Processed Quickly]
Achieve [Transactions Processed Quickly]
Achieve [Fraudulent Patterns Tested Against Stored Transactions]
Number of Txn Checks Exceeds Alert Generator Processing Speed

Figure 7.12: Case Study 3, Scalability obstacle to Alert Generator goals.

Note that the obstacle Number of Txn Checks Exceeds Alert Generator Processing Speed can be defined more precisely for each of the goals it obstructs. For example, sub-obstacles to the goal Achieve[Batch Processed Quickly] can be defined with respect to the different application domain characteristics, such as Batch Size Exceeds Alert Generator Processing Speed, Number of Entities Exceeds Alert Generator Processing Speed, and Number of Patterns Exceeds Alert Generator Processing Speed. A list of obstacles and sub-obstacles for each agent is shown in Tables 7.3 and 7.4.

[5] The real classification of pattern types have been omitted to protect Searchspace's intellectual property.

Scalability obstacles to goals under the responsibility of the Data Storage

The agent Data Storage is responsible for three of the goals in the standard goal model: Achieve[Incoming Transactions Stored], Achieve[Alerts Stored], and Achieve[Fraudulent Patterns Stored], as shown in the agent load diagram in Figure 7.13.

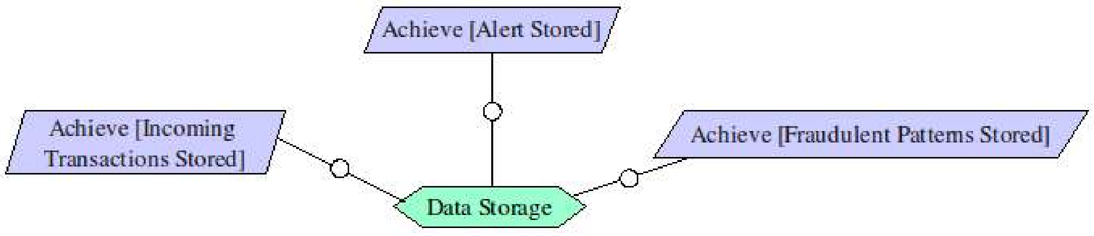

Achieve [Alert Stored]
Achieve [Incoming Transactions Stored]
Achieve [Fraudulent Patterns Stored]
Data Storage

Figure 7.13: Case Study 3, Data Storage agent responsibilities.

Since all these goals are concerned with the storage of data, the agent *capacity* is defined in terms of the amount of data the Data Storage can store. The *load* imposed on this agent (also defined in terms of the amount of data) depends on different application domain characteristics that have been overlooked in the standard goal model: number of fraudulent patterns, size of fraudulent patterns, number of transactions per second, number of transactions in a batch, size of transactions, number of new business entities per second, number of new business entities in a batch, size of business entities, number of alerts, and size of alerts. All these characteristics, but the size of alerts, will scale over expected ranges. The size of alerts is fixed. [6]

Therefore, the load on each goal can be considered infinite, inducing the scalability obstacle Amount of Data Exceeds Data Storage Space, as show in Figure 7.14. As in the previous example, this obstacle can be further refined into sub-obstacles to each of the goals, such as Number of Transactions Exceeds Data Storage Space, Number of Business Entities Exceeds Data Storage Space, Number of Alerts Exceeds Data Storage Space, and Number of Patterns Exceeds Data Storage Space. These obstacle and sub-obstacles are listed in Tables 7.3 and 7.4.

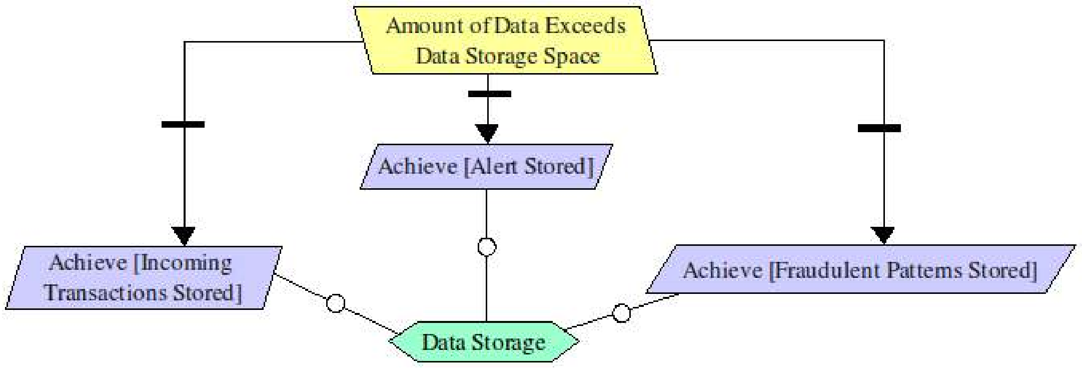

Amount of Data Exceeds Data Storage Space
Achieve [Alert Stored]
Achieve [Incoming Transactions Stored]
Achieve [Fraudulent Patterns Stored]
Data Storage

Figure 7.14: Case Study 3, Data Storage agent responsibilities.

[6]The size of an alert is considered part of the problem domain because the new Alert Generator has to be compatible with other subsystems of Searchspace, such as the Alert Resolution Workflow.

Scalability obstacles to goals under the responsibility of the Bank Staff

As a last example, consider the agent Bank Staff, responsible for the goal Achieve[Alerts Investigated and Acted Upon]. The agent *capacity*, in this case, is the number of alerts processed in a day by the Bank Staff. The *load* imposed on this agent (also defined in terms of the number of alerts processed in a day) depends on the number of alerts and the number of case-worthy alerts, which requires more investigation and resolution time. Both characteristics have been overlooked in the standard goal model. Therefore, the load on the goal Achieve[Alerts Investigated and Acted Upon] can be considered infinite, inducing the scalability obstacle Number of Alerts Exceeds Bank Staff Processing Speed, as illustrated in the Figure 7.15.

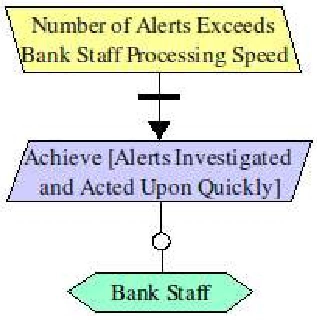

Number of Alerts Exceeds Bank Staff Processing Speed
Achieve [Alerts Investigated and Acted Upon Quickly]
Bank Staff

Figure 7.15: Case Study 3, Bank staff agent responsibilities.

Scalability obstacles in the standard goal model

Repeating the process for the other agents in the standard goal model, the set of scalability obstacles identified is listed in Tables 7.3 and 7.4 below. Note that we chose not to generate scalability obstacles to the expectations Achieve[Transactions Delivered Periodically in Batches] and Achieve[Transactions Delivered Continuously]. Both goals are under the responsibility of the environment agent Bank System, whose scaling characteristics defining the load on these goals are unknown by Searchspace. [7] Note that this is different from a scalability obstacle to the goal Achieve[Transactions Processed Quickly], caused by the Bank System delivering a number of transactions that falls outside to the range expected by Searchspace.

[7]Sometimes, however, it might be interesting to assume the existence such obstacles, as their potential occurrence may have consequences for system being analyzed. By anticipating these consequences, the system can be designed to provide the best service if an external agent in unable to satisfy its goals.

Agent	Scalability Obstacle	Goal
Alert Generator	Number of Txn Checks Exceeds Alert Generator Processing Speed	Achieve [Batch Processed Quickly]
		Achieve [Transactions Processed Quickly]
		Achieve [Fraudulent Patterns Tested Against Stored Transactions]
Data Storage	Amount of Data Exceeds Data Storage Space	Achieve [Incoming Transactions Stored]
		Achieve [Alerts Stored]
		Achieve [Fraudulent patterns Stored]
Bank Staff	Number of Alerts Exceeds Bank Staff Processing Speed	Achieve [Alerts Investigated and Acted Upon]
Searchspace Staff	Number of Fraudulent Patterns Exceeds Searchspace Staff Encoding Speed	Achieve [Known Fraudulent Patterns Encoded]

Table 7.3: Case Study 3, Scalability obstacles.

Scalability Obstacle	Refined Scalability Obstacle
Number of Txn Checks Exceeds Alert Generator Processing Speed	Batch Size Exceeds Alert Generator Processing Speed
	Number of Incoming Transactions Exceeds Alert Generator Processing Speed
	Number of Stored Transactions Exceeds Alert Generator Processing Speed
	Number of Patterns Exceeds Alert Generator Processing Speed
	Number of New Patterns Exceeds Alert Generator Processing Speed
	Number of Entities Exceeds Alert Generator Processing Speed
Amount of Data Exceeds Data Storage Space	Number of Transactions Exceeds Data Storage Space
	Number of Business Entities Data Exceeds Data Storage Space
	Number of Alerts Exceeds Data Storage Space
	Number of Patterns Exceeds Data Storage Space

Table 7.4: Case Study 3, Refined scalability obstacles.

Assessing Obstacles

Obstacles assessment consists of estimating the likelihood and criticality of scalability obstacles. As discussed in the previous step, since a number of application domain characteristics have been overlooked in the standard goal model, the load on the goals impacted by these characteristics can be considered infinite. Therefore, strictly speaking, the likelihood of the scalability obstacles *in this first assessment* can be classified as ‘almost certain’. For this reason, we show in Table 7.5 only the estimated criticality of the identified scalability obstacles.

Scalability Obstacle	Criticality
Number of Txn Checks Exceeds Alert Generator Processing Speed	high
Amount of Data Exceeds Data Storage Space	high
Number of Alerts Exceeds Bank Staff Processing Speed	moderate
Number of Fraudulent Patterns Exceeds Searchspace Staff Encoding Speed	low

Table 7.5: Case Study 3, Criticality of scalability obstacles.

Scalability goal-obstacle analysis is an iterative process. The loop identify-assess-resolve is repeated until all remaining obstacles are considered acceptable. When application domain characteristics have been overlooked, the tactic *introduce scaling assumption* is normally applied to establish the possible range of values for these characteristics, as it will be demonstrated in the next step. Once the overlooked application domain characteristics have been bounded, the scalability obstacles must be re-assessed. At this point, it may also be interesting to assess the likelihood of scalability sub-obstacles individually. Such assessment may require further analysis—for instance, back-of-the-envelope calculations, modeling or prototyping. For the purposes of the case study, we simply assumed the obstacles were likely to occur and applied the resolution tactics described in Chapter 5.

Resolving Obstacles

The generation of resolutions to scalability obstacles was performed by iterating through the catalog of resolution tactics and checking whether they could be instantiated for the obstacles at hand. Some of the resolutions we describe in this step were, in fact, used by Searchspace; others are simply alternative solutions to scalability obstacles.

Resolutions to the scalability obstacle Number of Txn Checks Exceeds Alert Generator Processing Speed

As shown in step 1, this obstacle can be defined more precisely for each of the goals it obstructs. Take, for example, the obstacle Batch Size Exceeds Alert Generator Processing Speed that obstructs the goal Achieve[Batch Processed Quickly].

The first resolution tactic instantiated to the obstacle Batch Size Exceeds Alert Generator Processing Speed was the *introduce scaling assumption*, which defines scaling assumptions on each of the application domain characteristics that had been overlooked in the standard goal model, and modifies the specification of the obstructed goals in such a way that they only need to be satisfied under the conditions expressed in the scaling assumptions. Two forms of describing a scaling assumption on the number of transactions in a batch were given in the Assumptions 5.1 and 5.2 of Chapter 5. We next exemplify a scaling assumption on the number of fraudulent patterns. Other assumptions are defined in a similar fashion.

Assumption 7.1

Assumption Expected Number of Fraudulent Patterns Variation
Category Scalability
Definition The IEF can have a number of different products running concurrently, each with a maximum of 200 fraudulent patterns. The overall maximum number of fraudulent patterns the system is expected to deal with is 1000. This assumption is valid for the next three years.

Considering that the scalability obstacle Batch Size Exceeds Alert Generator Processing Speed is still likely to occur, we examine the other resolution tactics. In Chapter 5, we have already described resolutions for this scalability obstacle using the tactics *Goal substitution*, *Adapt agent capacity at runtime according to load*, *Set agent capacity upfront to worst-case load*, *Strengthen scaling assumption*, *Relax real-time requirement*, and *Relax required level of satisfaction*.

An alternative resolution instantiates the tactic *Limit goal load according to fixed agent capacity*. In this resolution, the IEF receives transactions continuously from the Bank System and accumulate them in batches, never allowing the batch to grow over the maximum expected batch size the Alert Generator can handle. This resolution is illustrated by Figure 7.16. The dynamic counterpart of this resolution would be to adjust the size of the batch at runtime according to the available capacity of the Alert Generator. That tactic is the *Limit goal load according to varying agent capacity*.

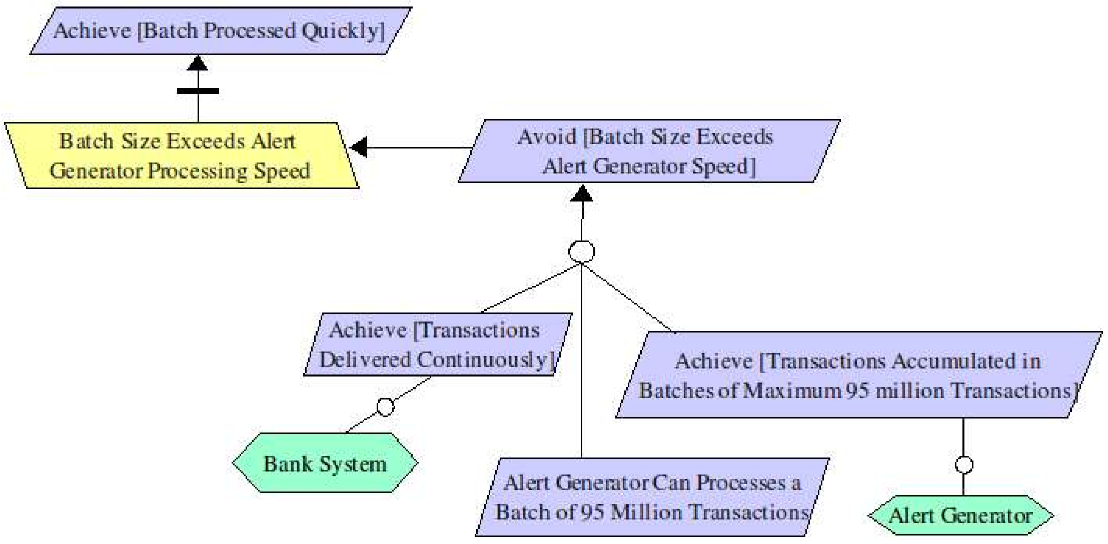

Achieve [Batch Processed Quickly]
Batch Size Exceeds Alert Generator Processing Speed
Avoid [Batch Size Exceeds Alert Generator Speed]
Achieve [Transactions Delivered Continuously]
Achieve [Transactions Accumulated in Batches of Maximum 95 million Transactions]
Bank System
Alert Generator Can Processes a Batch of 95 Million Transactions
Alert Generator

Figure 7.16: Case Study 3, Applying tactic *Limit goal load according to fixed agent capacity*.

The tactic *Transfer goal to non-overloaded agent* can also be applied to reduce the probability of occurrence of the obstacle Number of Txn Checks Exceeds Alert Generator Processing Speed. A solution actually adopted by Searchspace to reduce the load on the Alert Generator was to create a separate environment for testing new fraudulent patterns. More precisely, the responsibility for the goal Achieve [Fraudulent Patterns Tested Against Stored Transactions] is transferred to another agent, named Sandbox. This agent runs in a separate environment and has as its only responsibility the testing of new fraudulent patterns. The goal model is then refined as in Figure 7.17.

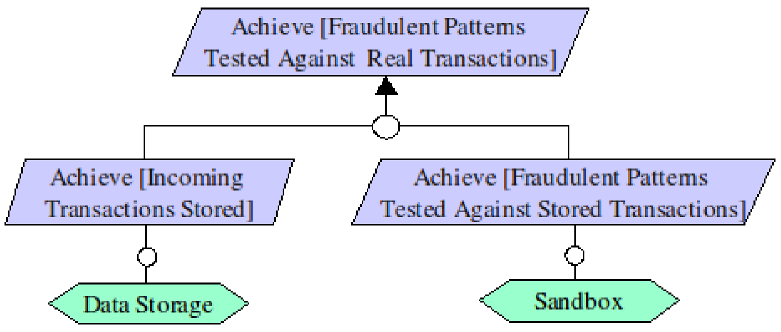

Achieve [Fraudulent Patterns Tested Against Real Transactions]
Achieve [Incoming Transactions Stored]
Achieve [Fraudulent Patterns Tested Against Stored Transactions]
Data Storage
Sandbox

Figure 7.17: Case Study 3, Applying tactic *Transfer goal to non-overloaded agent*.

The tactic *Split goal load among multiple agents* could also be applied to mitigate the obstacle Number of Txn Checks Exceeds Alert Generator Processing Speed. For example, the load could be *split by case*, where different agents are responsible for different types of transactions and frauds.

Finally, reaching the end of the catalog, the tactic *Goal restoration* can be applied to store the transactions that could not be processed within the expected time window, re-submitting them to the Alert Generator when it is less busy.

In this example, we have shown that alternative resolutions can be generated by systematically iterating through the catalog of obstacle resolution tactics described in Chapter 5. Note that, with the support the catalog, one can generate a number of alternative resolutions that are more creative than the usually adopted solution of increasing the system capacity. In next examples, we list only a few of the possible resolutions that have been obtained in a analogous fashion.

Resolving the scalability obstacle Amount of Data Exceeds Data Storage Space

As in the previous example, the first resolution tactic instantiated to mitigate the obstacle above was the *introduce scaling assumption*. The tactic was used, for example, to refine the goal Achieve[Alerts Stored] into scaling assumptions for the number and size of alerts and a refined specification of the obstructed goal such that it only needs to be satisfied under the conditions of these assumptions. The size of alerts is known from previous IEF versions and the number of alerts can be estimated through domain knowledge, such as the percentage of transactions that are fraudulent, and Searchspace's history in terms of the rate of false alerts and missed fraud. This resolution is illustrated in Figure 7.18.

This same tactic was applied to refine the goals Achieve[Incoming Transactions Stored] and Achieve[Fraudulent Patterns Stored]. The expected number of fraudulent patterns has already been shown in Assumption 7.1. The size of fraudulent patterns as well as the number and size of transactions and business entities are all known from previous versions of the IEF.

Considering that the scalability obstacle Amount of Data Exceeds Data Storage Space is likely to occur for the ranges of values established with the tactic above, different tactics can be applied to mitigate this obstacle. For example, by applying the tactic *Set agent capacity upfront to worst-case load*, one can estimate the maximum amount of space required to store incoming transactions, business entities, alerts,

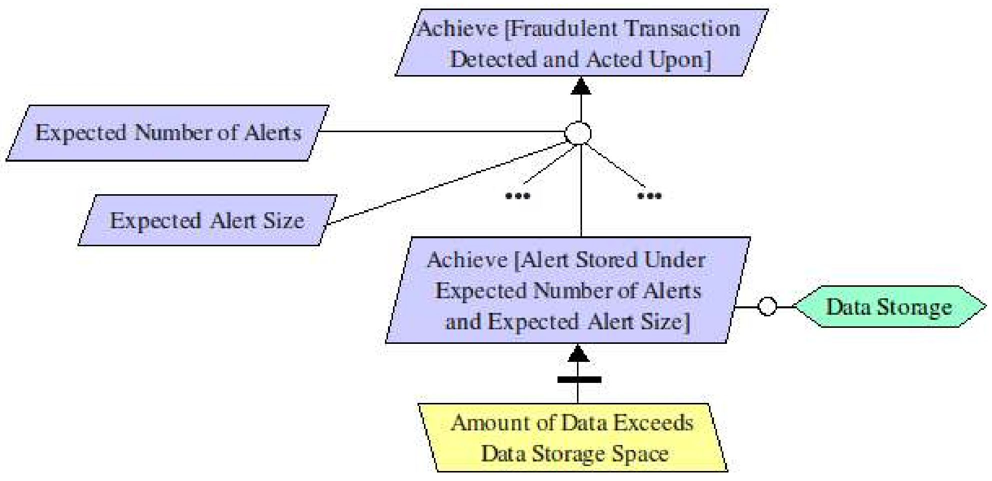

Achieve [Fraudulent Transaction Detected and Acted Upon]
Expected Number of Alerts
Expected Alert Size
Achieve [Alert Stored Under Expected Number of Alerts and Expected Alert Size]
Data Storage
Amount of Data Exceeds Data Storage Space

Figure 7.18: Case Study 3, Alerts stored.

and fraudulent patterns based on the scaling assumptions for number and size of these characteristics; and create a goal that requires the Data Storage to have sufficient space upfront to support this maximum load. Alternatively, one can apply the tactic *Adapt agent capacity at runtime according to load*, creating a goal to predict the amount of data to be stored in the near future, and another goal to add more space to the Data Storage agent accordingly. These resolutions are shown in Figure 7.19.

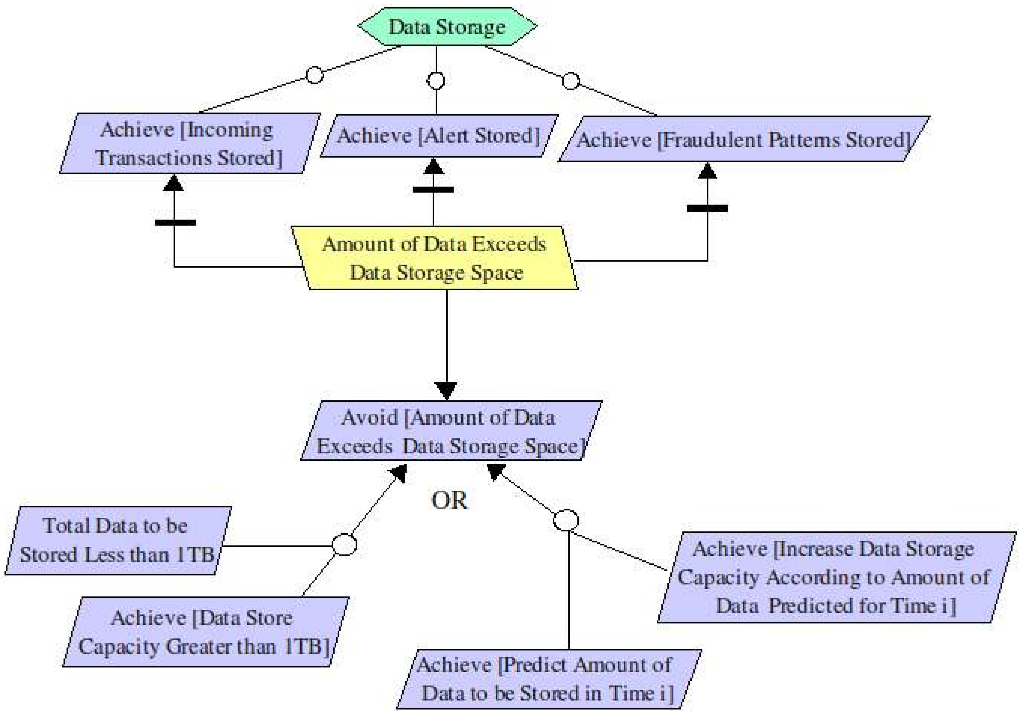

Data Storage
Achieve [Incoming Transactions Stored]
Achieve [Alert Stored]
Achieve [Fraudulent Patterns Stored]
Amount of Data Exceeds Data Storage Space
Avoid [Amount of Data Exceeds Data Storage Space]
OR
Total Data to be Stored Less than 1TB
Achieve [Data Store Capacity Greater than 1TB]
Achieve [Increase Data Storage Capacity According to Amount of Data Predicted for Time i]
Achieve [Predict Amount of Data to be Stored in Time i]

Figure 7.19: Case Study 3, Applying tactics *Set agent capacity upfront to worst-case load* and *Adapt agent capacity at runtime according to load*.

The obstacle Amount of Data Exceeds Data Storage Space can also be mitigated by applying the tactic *Goal substitution*. In fact, fraudulent patterns can normally be tested with one month's worth of transactions. Therefore, it is sufficient to store incoming transactions for 30 days only. Similarly, by law, information about fraudulent transactions only needs to be stored for a limited amount of time. The obstructed goals are therefore replaced by Goals 7.4 and 7.5.

Goal 7.4

Goal Achieve [Incoming Transactions Stored for a Limited Time Under Transaction Volume Variation]
Category Scalability
Definition All incoming transactions should be stored for 30 days, provided that the number and size of transactions do not exceed the bounds stated in the scaling assumptions '*Expected Transactions Size Variation* and *Expected Number of Transactions Variation*.[a]
Responsibility: Data Storage

[a]Assumptions describing the expected variation in the size of transactions and number of incoming transactions.

Goal 7.5

Goal Achieve [Alerts Stored for a Limited Time Under Alert Volume Variation]
Category Compliance and Scalability
Definition All alerts generated should be stored for 180 days, provided that the number and size of alerts do not exceed the bounds stated in the scaling assumptions *Expected Alert Size Variation* and *Expected Number of Alerts Variation*.[a]
Responsibility Data Storage

[a]Assumptions describing the expected variation in the size of alerts and number of alerts generated per day.

If limiting the time that the alert data is stored does not resolve the scalability obstacles, an option is to apply the tactic *Goal substitution* again. Note that a portion of the alerts generated by the IEF are, in fact, false positives (alerts for transactions that are not fraudulent). The need for storage space can be reduced by replacing the goal Achieve[Alerts Stored for a Limited Time Under Alert Volume Variation] by another goal Achieve[Only Case-worthy Alerts Stored for a Limited Time Under Alert Volume Variation], requiring that only the alerts whose fraud have been confirmed are stored. The new goal is specified as follows:

Goal 7.6

Goal Achieve [Case-worthy Alerts Stored for a Limited Time Under Alert Volume Variation]
Category Compliance and Scalability
Definition All alerts whose fraud has been confirmed by the bank staff should be stored for 180 days, provided that the number and size of alerts does not exceed the bounds stated in the scaling assumptions *Expected Alert Size Variation* and *Expected Number of Alerts Variation*
Responsibility: Data Storage

Resolving the scalability obstacle Number of Alerts Exceeds Bank Staff Processing Speed

As with the other two examples, the tactic *introduce scaling assumption* was used to bound the number of alerts and the number of worth-case alerts, as shown in the Figure 7.20. This obstacle can also be mitigated by applying the tactic *Split goal load into subtasks*, which refines the goal Achieve[Alerts Investigated and Acted Upon Quickly] into two separate goals Achieve[Alerts Investigated Quickly] and Achieve[Fraudulent

Transaction acted upon quickly], each assigned to an agent with a specialized role. The former goal is assigned to the agent Fraud Investigator, while the latter is assigned to the agent Collector. This division is, in fact, adopted by some of Searchspace's customers. Figure 7.21 models the application of this tactic.

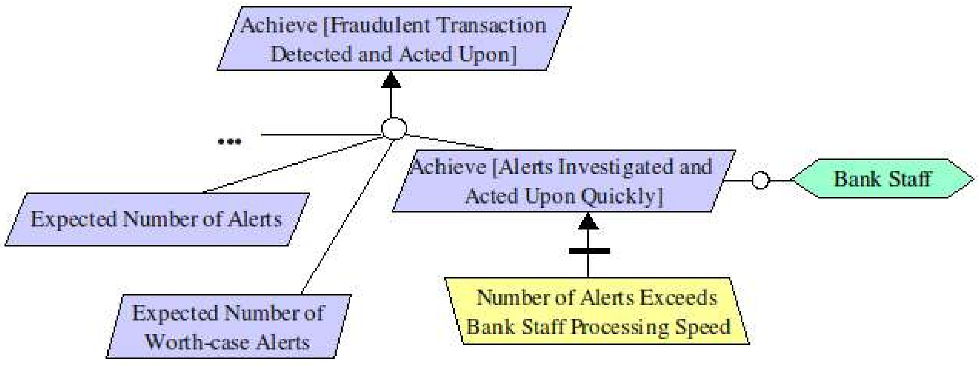

Achieve [Fraudulent Transaction Detected and Acted Upon]
...
Achieve [Alerts Investigated and Acted Upon Quickly]
Bank Staff
Expected Number of Alerts
Number of Alerts Exceeds Bank Staff Processing Speed
Expected Number of Worth-case Alerts

Figure 7.20: Case Study 3, Alerts investigated.

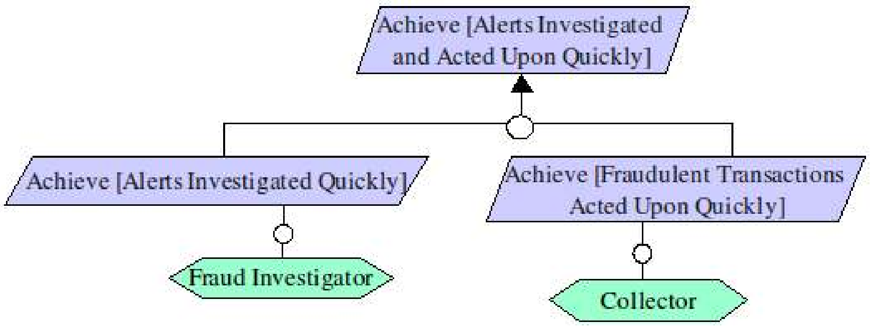

Achieve [Alerts Investigated and Acted Upon Quickly]
Achieve [Alerts Investigated Quickly]
Achieve [Fraudulent Transactions Acted Upon Quickly]
Fraud Investigator
Collector

Figure 7.21: Case Study 3, Applying tactic *Split goal load into subtasks*.

If despite all the above measures, the number of alerts still exceeds the fraud investigator processing speed, the tactics *Obstacle mitigation* and *Goal restoration* can be applied. It is possible, for example, to block every account associated with more than one alert automatically, ensuring the ancestor goal Avoid[Loss Due to Fraudulent Transaction] (obstacle mitigation); and postponing their investigation to a quieter time (goal restoration). In this case, the goal model would be refined as in Figure 7.22.

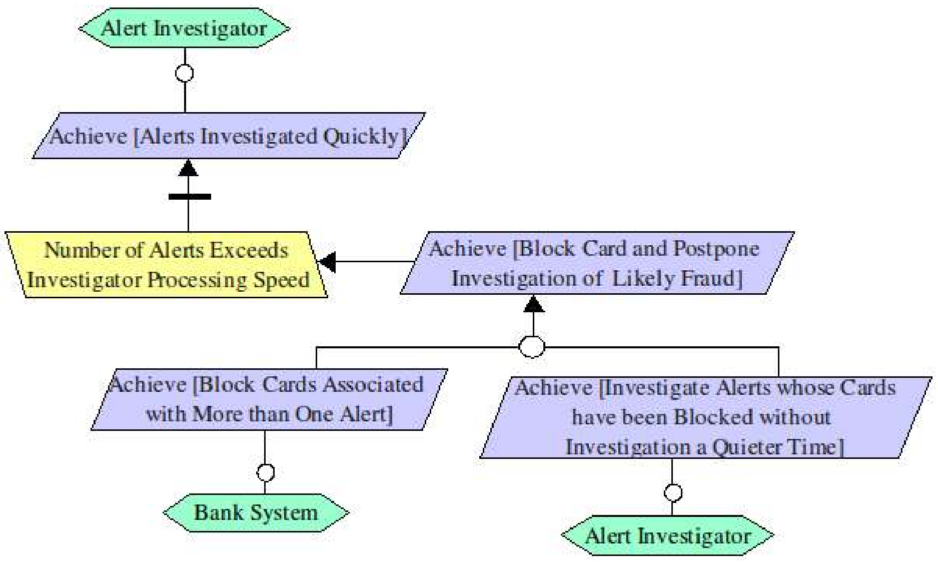

Alert Investigator
Achieve [Alerts Investigated Quickly]
Number of Alerts Exceeds Investigator Processing Speed
Achieve [Block Card and Postpone Investigation of Likely Fraud]
Achieve [Block Cards Associated with More than One Alert]
Achieve [Investigate Alerts whose Cards have been Blocked without Investigation a Quieter Time]
Bank System
Alert Investigator

Figure 7.22: Case Study 3, Applying tactics *Obstacle mitigation* and *Goal restoration.*

Step 3: Instantiation of variables and functions required for scalability analysis from selected goals

We next describe the instantiation of the framework for two types of scalability analysis, one that aims to understand the scaling trend of a particular goal and another that looks at an agent's ability to support the load imposed by its goals. However, first we review in the box below the mapping between the goal model and the scalability analysis variables and functions described in Chapter 6:

Independent variables: Application domain variables are derived from *scaling and non-scaling domain assumptions*. Variables from the system design are derived from characteristics of the *agents responsible for the goals*.

Dependent variables: Derived from *quality variables of scalability goals* and from *characteristics that measure the usage of an agent's capacity*.

Preference functions: Derived from *objective functions in scalability goals*.

Utility function: If using objective weighting, the utility function can be derived from the weights assigned their originating goals by a requirements prioritization technique.

The scalability goal-obstacle analysis of a KAOS goal model, with the objective to instantiate the variables and functions for the analysis, can be made at different points of the development lifecycle. In the case of the IEF, the requirements engineering activity was performed in parallel with the development of the third version of the system. As we progressed with the construction of the model, so did the IEF developers with their own activities. For this reason, the developers already had in mind a rough idea of the intended design. We could take advantage of this knowledge when instantiating variables and functions for a scalability analysis. Note that the intended design, in reality, is describing the characteristics of the software agent responsible for the goal. There are also a number of standard measures the IEF developers were interested in collecting for the diagnosis of eventual problems. These measures characterized not only the usage of the agent's capacity (e.g., memory usage, disk usage, CPU usage), but also some performance measures that may help to identify bottlenecks and causes of problems (e.g., disk I/O, network I/O, database read/write speed).

Scalability analysis: Analyzing the Trends of a Particular Goal

Chapter 6 suggests that in order to instantiate systematically the variables and functions for this kind of analysis, one should select (1) the scalability goal whose trend should be characterized; (2) all scaling and non-scaling assumptions related to that goal; and (3) all agents assigned to that goal.

Take, for example, the goal Achieve [Transactions Processed Quickly] (Goal 7.2). In the refined model, this goal is associated with scaling assumptions specifying the ranges of the following characteristics: number of transactions per second, number of distinct business entities per second, and number of fraudulent patterns. The agent assigned to that goal is the Alert Generator. Developers intended to implement the Alert Generator as a software component running in the IEF platform. In their intended design, each fraudulent pattern is implemented as a rule that is checked against incoming transactions. Transactions may be checked in parallel by separate threads in the Alert Generator. Most of the rules may run in parallel, but some rules depend on the results of previous rules, so they have to run in a particular order. Rules also have different types, which correspond to the types of fraudulent patterns. For this reason, not only the number or type of rules (or fraudulent patterns) matter, but also the percentage of rules that depend on others, and the dependency depth of these rules. Finally, system designers hoped to improve scalability by running a number of Alert Generators in a computer cluster. The specification of these machines and the network also have an impact on the satisfaction of the goal.

From the goal model and the description of the intended design, the variables and functions were instantiated as shown below. Note that we have included dependent variables that measure resource usage (i.e., characteristics that determine the usage of the Alert Generator's capacity). These variables are used to explain the scaling trends observed in the scalability tests.

Scalability Question:

How does the processing time of incoming transactions on the IEF vary with the number of transactions per second, distinct business entities per second, and fraudulent patterns?

Quality goal:

Performance and scalability goal: *Achieve[Transactions Processed Quickly]*

Independent variables:

Scaling variables:

Number of transactions per second

Number of distinct business entities per second

Number of fraudulent patterns (rules)

Number of Alert Generators in cluster

Number of threads in Alert Generators

Non-scaling variables:

Type of fraudulent pattern (rules)

Percentage of dependent rules

Average rule dependency depth

Network bandwidth

Machine CPU speed

Machine RAM

Disk speed

Dependent variables:

Average transaction processing time

Average CPU usage

Average Network throughput

Maximum Memory usage

Average Disk I/O

Preference function:[8]

$$\text{pref(processingTime)} = \begin{cases} 1, & \text{if Pr(processingTime} \leq \text{0.5ms)} \geq \text{0.8 and Pr(processingTime} > \text{2ms) = 0} \\ \dfrac{\text{Pr(processingTime} \leq \text{0.5ms) - 0.8}}{0.1}, & \text{otherwise.} \end{cases} \tag{7.9}$$

Scalability Analysis: Analyzing an Agent's Ability to Satisfy a Scalability Goal

In order to instantiate systematically the variables and functions for this kind of analysis, one should select (1) the scalability goal we are interested in; (2) the agent whose ability to satisfy that goal we aim to analyze; (3) all goals that might also require the resources of that agent; (4) the scaling and non-scaling assumptions associated with all selected goals; and (5) possibly some cost goal if we want to investigate a scalability strategy in which the agent's capacity is varied with the load.

From the goal model and the intended design, the variables and functions were instantiated as follows:

Scalability Question:

Can the IEF store incoming transactions for 180 days given a varying number and size of transactions, business entities, alerts and fraudulent patterns?

Quality goal:

Achieve [Incoming Transactions Stored for a Limited Time Under Transaction Volume variation]

[8]There are no goals associated with the other dependent variables (for resource usage), therefore only the preference for processing time is defined.

Independent variables:

Scaling variables:

Number of transactions per second

Number of transactions in batch

Size of transactions

Number of rules (or fraudulent patterns)

Size of rules

Number of business entities

Number of alerts

Size of business entities

Number of threads

Non-scaling variables:

Size of alerts

Database tuning parameters

Remote vs local disk

Network bandwidth

Dependent variables:

Maximum Disk usage

Average CPU usage

Average Disk I/O

Preference function: [9]

$$\text{pref(usedSpace)} = \begin{cases} -10.000, & \text{if availableSpace - usedSpace = 0} \\ 1, & \text{otherwise.} \end{cases} \quad (7.10)$$

Step 4: Considerations and Conclusions

Unlike a retrospective study of a fully implemented system with well understood requirements, this project involved a system under development, with all the complexities normally present on a requirements engineering exercise. The actual system was in the requirements specification stage, with many conflicting views and uncertainties regarding the system's goals. Stakeholders were busy with their own daily tasks and reluctant to provide estimates about the application domain in the future. There were multiple, incomplete and, sometimes, conflicting sources of information.

The requirements elicitation and construction of the goal model was performed by the PhD candidate.[10] She had worked for three and a half years as a developer at Searchspace and, therefore, had some domain knowledge. She had no professional experience in requirements engineering. KAOS was learned through technical papers and discussions about the models with an expert researcher on the area. The requirements engineering activities took approximately 30 days, including the interviews, brain-

[9]There are no goals associated with the other dependent variables, therefore only the preference for disk usage is defined.

[10]Threats to validity associated with this choice are discussed in Section 7.5.6.

stormings, data characterization, document review, modeling, and exploration of the new techniques for elaborating the scalability requirements described in this thesis. A breakdown of these times is shown in Table 7.6.

Time	**Activity**
2 days	The analyst built an initial high-level model with her own domain knowledge to serve as a starting point for discussions.
25 days	The analyst built the standard goal model and applied goal-obstacle analysis to identify scalability obstacles and scaling assumptions.
5 days	The analyst explored alternative resolutions to identified obstacles and refined the model.

Table 7.6: Case Study 3, Recorded times of activities.

Validation of the Hypotheses

Ideally, for validation of the hypothesis, the resulting goal model would be compared with what, in reality, happened to the system during production. The objective of such evaluation would be to assess whether the technique has described the relevant scaling characteristics of the application domain, the correct levels of goal satisfaction under the range of values for these characteristics, and the obstacles to these goals. Nevertheless, such a comparison is not viable within the time frame of a PhD research programme. Evaluation, instead, was performed by demonstrating the application of the proposed techniques on a large-scale industrial system, with the support of rational argumentation. Comparison of the resulting goal model against the original set of requirements for the system and against the goal model developed with the standard KAOS technique is also used for evaluation.

Similarly, this case study is concerned with the derivation of scalability analysis variables and functions from a goal model. Evaluation of the case study in that respect is also a combination of rational argumentation and the demonstration of the technique in a real-world system. In the particular case of the IEF, a limited comparison can be made against the variables and functions used in the first two case studies.

The hypotheses have been validated as follows:

Validation of hypothesis H.3.1: This hypothesis states that the techniques can be applied to large-scale, industrial projects. Validating this hypothesis requires evidence that the technique has been applied to a number of real-world systems within an industrial context, with all the complexities normally involved in a requirements engineering activity.

The evidence provided by the case study is as follows:

- The set of scaling assumptions, scalability goals, scalability obstacles and resolutions described in Section 7.5.5; and
- the recorded times of the different activities listed in Table 7.6 demonstrating that the technique was successfully applied in a large-scale, industrial project.

Due to the complexity of the task, the technique could only be applied to a single system. Therefore Hypothesis H.3.1 has been only partially validated by this case study. Difficulties of the exercise, limitations of the technique, and benefits to Searchspace are discussed in Section 7.5.6.

Validation of hypothesis H.3.2: This hypothesis states that the technique should allow one to identify systematically important scaling and non-scaling characteristics of the application domain that may be missed if no systematic technique is applied. Validating this hypothesis requires evidence that the technique unveiled scaling and non-scaling assumptions on application domain characteristics that are likely to have been overlooked otherwise. A limited comparison of this model can be done against the original requirements specification document and standard KAOS goal model.

The evidence provided by the case study is as following:

- The specification of scaling and non-scaling characteristics of the IEF's environment specified in domain assumptions. Assumption 7.1 is an example of a scaling assumption. Goal 7.4 is an example of a scalability goal that relies on this assumption;
- A set of 13 application domain characteristics that may affect the satisfaction of the IEF goals, uncovered by the technique in the portion of the goal model described in this chapter. These characteristics and the required level of goal satisfaction with respect to them have been overlooked in the original set of business requirements for the IEF and in the standard KAOS goal model. They are: number of transactions in a batch, number of distinct business entities in a batch, number of transactions per second, number of distinct business entities per second, number of transactions in testing data, number of business entities in testing data, number of fraudulent patterns, number of new fraudulent patterns on any given day, size of fraudulent patterns, size of transactions, size of business entities, size of alerts, and number of case-worthy alerts.

The scaling domain assumptions have been uncovered by iterating through all the scalability obstacles in the goal model and applying the tactic *introduce scaling assumption* to the ones that were caused by a unbounded range of some application domain characteristic; which demonstrates the systematic nature of our technique.

In addition to the ranges of application domain characteristics, the quality requirements and their varying levels of goal satisfaction had been almost completely overlooked in the original business requirements specification for the system. The comparisons used to validate this hypothesis have their limitations, as discussed in Section 7.5.6. We next use rational argumentation to complement the evidence presented above:

The original requirements specification document was built without a systematic technique, yet it had been approved to drive the design and implementation of the system. Our comparison, therefore, does demonstrate that application domain characteristics have been overlooked. Additionally, in the standard KAOS model, goals such as Achieve [Batch Processed Quickly], Achieve [Incoming Transactions Stored] and Achieve [Alerts investigated and acted upon quickly] did not identify the ranges of application domain characteristics that determined their load. Such an omission is not uncommon in goal models published in the computing literature. Take for example, the goal Achieve [Ambulance Intervention] (Goal 2.1, in Chapter 2) published for the London Ambulance Service (Letier and van Lamsweerde, 2004). This goal states that for every urgent call reporting an incident, an ambulance must arrive at the incident scene within 14 minutes, but does not state the number of simultaneous incidents that could occur at any given time, neither whether the level of satisfaction of this goal is allowed to vary with the number of simultaneous incidents.

We therefore argue that both comparisons, although limited, can be used as evidence to validate Hypothesis H.3.2.

Validation of hypothesis H.3.3: This hypothesis states that the technique should allow one to identify systematically a set of scalability risks with respect to all the goals and expectations in the model, some of which may be missed if no systematic technique is applied, and to link each scalability risk to the system goals it may impact. Validating this hypothesis requires evidence that all requirements and expectations in the goal model have been systematically examined for scalability obstacles, and that paths have been defined between the scalability obstacles and all goals they may affect.

The evidence provided by the case study is as following:

- The set of scalability obstacles listed in Tables 7.3 and 7.4, representing the risks associated with the satisfaction of scalability goals. Figures 7.13 and 7.15 are examples of scalability obstacles in the IEF. This set is considered comprehensive as it was obtained by iterating through all requirements and expectations associated with each agent in the model, as demonstrated in Section 7.5.5. These obstacles have been completely overlooked in the IEF's original requirements specification and in the standard goal model;
- The obstruction and goal refinement links that can be followed upwards to demonstrate the impact of the scalability obstacles in the system goals. For example, the obstacle Batch Size Exceeds Alert Generator Processing Speed obstructs not only the goal Achieve [Batch Processed Quickly], but also higher level goals such as Achieve [Transaction Matching Fraudulent Patterns Alerted], Achieve [Fraudulent Transactions Detected and Acted Upon], and Maintain [Banks Reputation].

The set of scalability obstacles was uncovered by iterating through all leaf quality goals in the model, and creating an obstacle to any goal that could not be satisfied because the agent responsible for that goal may not have sufficient capacity to support the load imposed on it. This iteration demonstrates the systematic nature of our technique.

As with the previous hypothesis, the comparisons used have their limitations. Rational argumentation complements the above evidence:

In the standard KAOS goal model produced in Step 1, the omission of application domain ranges implies a potentially infinite load and, consequently, (unrevealed) scalability obstacles. In order to demonstrate that such omission is not uncommon, we return to the London Ambulance Service example. The goal Achieve [Ambulance Intervention] is impossible to satisfy without some bound on the number of simultaneous incidents that could occur at any given time. As another example, take a goal-obstacle analysis for this same system (van Lamsweerde and Letier, 2000). In that work, the authors did not identify scalability obstacles, despite the fact that such obstacles had occurred in the real system and contributed to its failures. Consider the goal MobOrderSentToKnownStation requiring a mobilization order to be sent over a PSTN line to the station at which an allocated ambulance is waiting. We believe that our techniques would have identified the scalability obstacle Number of Mobilization Orders to be Sent Exceeds PSTN Lines Available, whereas the technique described by Lamsweerde and Letier only identified the more general obstacle MobOrderNotSentToKnownStation (van Lamsweerde and Letier, 2000).

The study therefore validates Hypothesis H.3.3.

Validation of hypothesis H.3.4: This hypothesis states that the technique should allow one to explore a wide range of alternative ways to deal with scalability risks systematically, and envision solutions that may be missed if no systematic technique is applied. Validating this hypothesis requires evidence that scalability obstacles have been systematically mitigated with the support of the resolution tactics described in Chapter 5, resulting in a range of non-obvious alternative resolution for scalability obstacles.

The evidence provided by the case study is:

- The set of obstacle resolution tactics described in Section 7.5.5. For example, 12 alternative resolutions for the obstacle Batch Size Exceeds Alert Generator Processing Speed were generated by iterating through the catalog of scalability resolution tactics and checking whether each of them could be instantiated for the scalability obstacle at hand.

This iteration through all the resolution tactics the catalog for each obstacle shows the systematic nature of our technique. By doing so, we also have generated more creative alternatives than simply increasing the capacity of the Alert Generator. One of the resolutions that had not been considered by Searchspace resulted from the application of the tactic *Limit goal load according to fixed agent capacity*, in which the IEF receives transactions continuously from the Bank System and accumulate them in batches, never allowing the batch to grow over the maximum expected batch size the Alert Generator can handle.

The study therefore validates Hypothesis H.3.4.

Validation of hypothesis H.3.5: This hypothesis states that the method in Chapter 6 should allow one to derive systematically, from a goal model, a set of variables and functions for a scalability analysis. Validating this hypothesis requires evidence of the derivation of variables representing characteristics of the application domain and system design, of measurable system qualities, and of functions representing the levels of satisfaction of goals under the variation of some of these variables. A limited comparison of these variables and functions can be done against the ones provided by the stakeholder for the first two case studies.

The evidence provided by the case study is:

- The instantiation of variables and functions for two kinds of scalability analysis, one concerned with the scaling trend in the processing time of transactions and another looking at the IEF capacity for storing an increasing amount of data;
- A set of variables and functions that have been overlooked or modeled incorrectly on the previous two case studies. For example, the stakeholder overlooked the number and type of fraudulent patterns, which also impacts the processing time of a transactional batch. Differences can also be found in the function for the batch processing time. In the first two case studies the preference stated that the higher the throughput, the better. This study, however, has shown that stakeholders ultimately are not interested in achieving the highest throughput possible. Instead, what they really want is to ensure that a transactional batch is processed within a given time window, as represented by the objective functions in the resulting goal graph.

The variables and functions have been selected by considering, one-by-one, all the goals, scaling assumptions and agents related to the two scalability concerns, which demonstrates the systematic nature of our technique. Nevertheless, as is discussed in Section 7.5.6, the selection of system design variables was performed in an ad-hoc manner. Therefore, our study partially validates Hypothesis H.3.5.

7.5.6 Evaluation

Critical evaluation of this study is divided into three parts: (1) threats to validity, (2) difficulties of the elaboration of scalability requirements and its benefits to Searchspace, and (3) evaluation of the KAOS extensions for elaborating scalability requirements.

Threats to Validity

The study just described has some limitations. The requirements engineering technique was applied by the PhD candidate, who may be biased to produce a conventional KAOS model with obstacles that can be mitigated by the proposed obstacle resolution tactics. Having the PhD candidate playing the role of analyst represents a threat to the *construct* and *external validity* of our study. Ideally, the standard goal model would have been produced by a requirements analyst experienced with KAOS, but unaware of our extensions to the technique or purpose of the study. Unfortunately, given the complexity of the task it was not possible to have a KAOS expert to produce a goal model of the IEF prior to the study. Therefore, although the PhD candidate did her best to produce a model with a similar style to the ones commonly found in the literature, there is no guarantee that such a model is not biased. This threat is partially reduced by a smaller study of the London Ambulance Service (LAS) system described in

Section 7.6.2. In this study, scalability obstacles elaborated with our technique are compared with a previously published set of obstacles resulted from the standard KAOS technique (van Lamsweerde and Letier, 2000). All scalability obstacles had been over overlooked in the original set of obstacles, and one of them (Number of Mobilization Orders to be Sent Exceeds PSTN Lines Available) indeed occurred in the real system and contributed to its failures.

Another threat to the *external validity* of our study is that, being an one-off study, it does not guarantee that our resolution tactics can be applied to real-world systems from different domains. However, due to time constraints it was infeasible to repeat the study in real-world systems from different domains. In order to address this problem, Section 7.6.2 illustrates scalability obstacles and resolution tactics on other systems described in the literature and in the industry interviews.

The case study also performed a few comparisons between the goal model resulted from our techniques and the original set of requirements for the IEF. Such a comparison may be seen as unfair, as the original IEF requirements were elaborated without a systematic requirements engineering technique. This is a threat to the *construct validity* of the study. However, despite being limited, the comparison is interesting because it shows the benefits of using our technique against the requirements that otherwise would have driven the design of a real-world system.

Finally, the exercise has identified a number of domain characteristics whose scaling range had not been previously estimated by Searchspace staff. Although, we have performed some data characterization ourselves, we did not have the time to estimate the range of all domain characteristics identified, which represents a threat to the *internal validity* of our study. In such cases, we just assumed the scalability obstacles were likely to occur and generated resolutions to them in order to illustrate the tactics.

Difficulties of the Requirements Engineering Exercise and Benefits to Searchspace

As with any requirements engineering exercise, human issues were a key contributor to the difficulties we experienced in applying KAOS to elaborate the scalability requirements of the IEF. Knowledge was spread across individuals, and their assumptions were a mix of "common sense" with peoples' experiences on different projects, documents were contradictory and incomplete, and many goals had no clear rationale other than being a reflection of older IEF versions. Some data characterization had been performed, but these were isolated attempts, whose results had not been widely circulated. Although our technique helps to identify what scalability-related information needs to be elicited during requirements engineering, we found the elicitation of measurable requirements particularly difficult. The varying characteristics of the customer base led to different assumptions to be considered and people found it difficult to provide definite estimates, forcing us on occasion to perform our own data characterization.

The requirements engineering exercise has been valuable to Searchspace. Project managers easily understood and accepted the basic concepts of KAOS and were particularly impressed by the impact of distinct projected values on leaf goals and the revelation of contradictory requirements. In fact, we modeled the new version of the IEF by invitation of the project manager, who had seen partial goal models we had produced previously for the IEF. KAOS provided rationale for low-level requirements and, in particular, it allowed a more precise and justifiable characterization of scaling ranges and objective functions. Some scaling bounds happened to be more flexible than originally stated, while others were found to be more rigid. One of the tools that enabled us to elaborate goals and estimate bounds more precisely was to get an explicit statement of all underlying assumptions about the application domain. The final model contained a number of goals that had been overlooked in Searchspace's initial requirements specification and replaced others that were idealized on that specification. The approach also helped to document and resolve the various stakeholder disagreements. Goal-obstacle analysis showed possible threats, and not only scalability related ones. Searchspace has used the goal model we produced to support the development of the new version of the IEF. Furthermore, the model can be used by them to assist a number of other activities, such as scalability analysis, derivation of test cases, and hardware sizing.

Evaluation of the KAOS Extensions for Elaborating Scalability Requirements

With respect to the technique itself, we have faced some difficulties in attempting to apply it to a real-world system. All these limitations are considered as future work in Chapter 8.

Modeling disagreements and uncertainties: The most significant difficulty in the process involved reaching agreements on the projected values for domain quantities in the scaling assumptions. The reason for this is that there are many uncertainties about future values, and different people had different opinions about them. The identification of scalability obstacles and description of scaling assumptions helped us uncover these disagreements (some of which had remain tacit until then), but we lacked a systematic approach to resolve them. Notably, the KAOS modeling language did not allow us to model disagreement about the content of a scaling assumption (other than by specifying separate assumptions which is an unsatisfactory solution), nor to model and reason about uncertainties concerning such assumptions. Such support might help the elicitation, negotiation, and analysis of scaling assumptions and scalability goals.

Likelihood of scalability obstacles: The capacity of the agent under development is typically unknown during requirements engineering; a purpose of the requirements engineering process is indeed to define what the required capacity of this agent should be. The assessment of the likelihood of scalability obstacles affecting this agent should therefore involve identifying how the obstacles' likelihood varies with the agent's required capacity. Such information together with the obstacle criticality and the costs associated with each alternative could then be used to decide which capacity should be deployed. Quantitative techniques such as those of Letier and Lamsweerde might help, but they require significant modeling effort (Letier and van Lamsweerde, 2004).

Context-sensitive agent load analysis: Defining the total load imposed by goals on an agent as the sum of the load imposed by the individual goals is a crude approximation. In reality, multiple goals will not all mobilize the agent's resource at the same time. In the IEF, goals dealing with alerts generation may mobilize the system's processing power during the night, whereas goals dealing with displaying the resulting alerts to fraud investigators mobilize the processing power during the day. Techniques are required to perform a quantitative agent load analysis that takes into consideration when goals consume the agents' capacity.

Selection of resolution strategy: Another important difficulty concerns the selection of preferred obstacle resolution strategies from among a set of alternatives. The systematic identification of scalability obstacles generated a wide range of meaningful alternative ways to improve the system scalability, which ranged from simplifying the logic of the alert generation rules to increasing hardware capacity. Alternative solutions have different short-term and long-term costs, as well as different impact on goal satisfaction (such as fraud detection speed and the rate of missed/false alerts). A quantitative cost/benefit analysis would be useful.

Load imposed by the system design: Some of the load imposed on agents is not captured by the goal model because it is determined by the system design. Take, for example, the storage capacity of the agent Alert Generator. In Figure 7.13, the goals using the Alert Generator's storage capacity are Achieve[Transactions Stored for a Limited Time], Achieve[Alerts Stored for a Limited Time] and Achieve[Fraudulent Patterns Stored]. However, as described earlier, developers intend to implement fraudulent patterns as rules that are checked against incoming transactions. Rules may depend on each other and, for this reason, rules results are stored, also consuming some of the Alert Generator's storage capacity. As Achieve[Rules Results Stored] is not a goal in the IEF goal model, it is not considered in the agent load analysis.

Adhoc selection of system design variables and diagnosis metrics: Recall that the scalability goal-obstacle analysis of a preexisting KAOS goal model can be performed at any point of the development lifecycle. In this case study, design knowledge was used to instantiate independent system design variables, albeit in an ad hoc manner. Furthermore, Searchspace developers showed interest in collecting standard metrics (e.g., memory usage, disk usage, disk I/O, network I/O) even when these metrics were not directly related to the quality variables and objective functions of the goals in question. They were also interested in collecting metrics on intermediate steps to satisfy a goal. For example, the goal Achieve [Transactions Processed Quickly] was described in terms of the quality variable processingTime; however, developers were interested in measuring the time spent in the different processing stages, such as the transaction parsing time, the rule execution time, the historical data retrieval time and so forth. Developers wished to collect these metrics for the diagnosis of eventual problems, such as the identification of bottlenecks if the targets for processing time were not achieved.

The scalability framework is still lacking systematic techniques for identifying independent system design variables and dependent "diagnosis" variables.

The experience also demonstrated some benefits of the techniques in a real-world setting:

KAOS advantages for scalability: Some of the well-known advantages of KAOS are particularly useful for elaborating scalability requirements. These include: uncovering assumptions about the application domain, providing rationale for goals, providing traceability for changing assumptions on application domain quantities, assigning responsibilities, and providing measurable quality variables and objective functions. These advantages have been discussed in Chapter 5.

Scalability goal-obstacle analysis: Goal-obstacle analysis has proved to be a systematic and intuitive way to reason about scalability during the early stages of software development, and to guide the specification of scaling assumptions and scalability goals. Furthermore, the study has demonstrated the application of the scalability obstacle resolution tactics in a real-world setting. In particular, the technique has the following values:

- **Comprehensive set of scalability obstacles**: this is guaranteed by the technique, which systematically examines all requirements and expectations associated with each system agent, searching for scalability obstacles.
- **Identification of non-obvious scaling characteristics**: The systematic specification of scalability obstacles led to the identification of several domain quantities that had been overlooked in our standard goal model. The specification of scaling assumptions for domain quantities led stakeholders to reconsider some of the initially envisaged requirements for the system, with some requirements being made weaker than initially specified and others stronger.
- **Accurate model of impact for scalability obstacles**: The impact of scalability obstacles on the system goals can be obtained by following upwards the obstruction and goal refinement links in the resulting goal model.
- **Range of alternative resolution for scalability obstacles**: A catalog of resolution tactics encodes design ideas into a coherent body of knowledge to make them more accessible and easy to apply, also allowing the exploration of a range of possible alternative resolutions to a problem.

Specifying scaling ranges in the application domain: In our previous experience with KAOS, we observed that there was a lack of systematic techniques for specifying scaling ranges in application domain characteristics. We also noticed that KAOS had no taxonomy for application domain assumptions or means to express assumptions that varied over time. With the case study, we showed that scaling assumptions clearly expressed all the relevant aspects of these ranges, as well as showed their impact on different IEF goals.

Modeling hardware-based scaling strategies: Another question raised in our first experience with KAOS was whether to model hardware-based scaling strategies[11] in goal trees. Our concern was that we would be inappropriately crossing the border between requirements and design in the goal model. We now see such strategies as being resolutions to scalability obstacles, which can be

[11]These are strategies where the system hardware capacity is increased to deal with the variation of quantities in the application domain (Weinstock and Goodenough, 2006).

expressed at the requirements level. For example, the obstacle Batch Size Exceeds Alert Generator Processing Speed can be resolved by maintaining the alert generator processing speed above what is needed to process the predicted maximal batch size. At design time, this tactic could be implemented either by increasing the hardware power or the number of machines in a cluster.

Assessing the combined load of goals on agents: Extending agent responsibility diagrams with scalability obstacles provided a lightweight assessment of the combined effect of the load imposed by different goals on a single agent. For example, the Alert Generator agent needs sufficient storage space to hold incoming transactions, alerts and fraudulent patterns.

7.5.7 Concluding Remarks

This section has described a study to evaluate the techniques for elaborating scalability requirements presented in Chapter 5 against a real-world system. The main points are:

- The study demonstrates the suitability of the technique to specify scalability requirements in goal models, and to derive variables and functions to be used in a scalability analysis.
- The study involved the PhD candidate and Searchspace staff. The study combined a number of activities, such as interviews, brainstorming, data characterization, and documents review to build the goal model.
- The study elaborated the requirements of the IEF, dealing with many of the complexities normally present in the real-world systems. For this reason, evaluation of hypotheses are a mix of demonstrating the application of the technique, comparison against the original IEF requirements and early case studies, and rational argumentation.

The case study contributed to the research by providing validation of the hypothesis, which demonstrates the suitability of our requirements engineering technique for a large real-world system. It has also benefited Searchspace, by identifying requirements and scaling characteristics that had been overlooked, replacing requirements that had been idealized, and highlighting inconsistencies and contradictions in Searchspace's original requirements specification documents.

The case study has also highlighted a number of shortcomings on our technique, such as the modeling of uncertainties in a goal model, the need for a quantitative assessment of scalability obstacles, and the need for systematic selection among scalability resolution strategies. All these shortcomings have been set as future work.

7.6 Further Examples and Informal Study

This section contains a number of smaller examples taken from the computing literature and from the industry interviews. These examples complement the cases studies, illustrating the application of the concepts and techniques described in this thesis to systems in a variety of application domains and with different designs.

7.6.1 Examples of Instantiating Framework Elements

In Chapter 4, scalability concerns from the Google search engine were recast according to the scalability framework elements. Other examples taken from the computing literature and industry interviews are given below. We describe each example in terms of quality goals, scalability question, independent and dependent variables, and preference functions, when possible. It is important to note that the level of detail with which the framework elements are instantiated is limited by the information available in the source documents from which the examples are drawn or from the stakeholders. Sometimes, assumptions had to be made.

Furthermore, although we argue that the instantiation of the scalability framework would have helped readers to understand what is being claimed by the authors of the papers, the main objective of this section is to show that different scalability concerns can be clearly and precisely described according the elements of our scalability framework.

Recast Examples from the Computing Literature

Distributed Systems

Jogalekar and Woodside address the question of a scalability measure for distributed heterogeneous service systems (Jogalekar and Woodside, 1997). They created a metric ψ that measures the return on investment for increasing the traffic handling capacity of the system, or for improved QoS. According to the authors, a distributed service system is considered scalable if $\psi \geq 1$.

We recast two examples from their paper according to the scalability framework. The first refers to quality goals and characteristics related to the scalability of distributed systems in general. The authors state that distributed systems normally aim to maximize throughput for a fixed response time and, conversely, minimize the response time for a fixed throughput. These objectives are instantiated in the scalability framework as the performance goals Maximize [Throughput for fixed response time] and Minimize [Response time for fixed throughput]. Another concern of the authors is whether the system can maintain or increase the power per invested dollar as it scales, where power is defined as the ratio between throughput and response time. This objective is represented in the scalability framework by the cost goal Avoid [Reducing the power per invested dollar]. In KAOS, these goals would be measured by the quality variables throughput, response time and cost in dollar. These variables are used to calculate the metric ψ, whose definition encompass the three goals above. The authors also identify the number of active clients as a scaling characteristic of the application domain that might affect the satisfaction of the above goals. In addition, they enumerate a number of characteristics of the system design that contribute to the satisfaction of these goals: the number of replicated databases, the number of replicated servers, the CPU speed, the disk speed, the communication links speed, the collocation/distribution of software tasks, the priority scheduling, the trading algorithm, and the name /directory service algorithm. The scalability framework is, therefore, instantiated as follows:

Quality goals:

Maximize [Throughput for Fixed Response Time]

Minimize [Response Time for Fixed Throughput]

Avoid [Reducing the Power per Invested Dollar]

Independent variables:

Scaling variables:

Number of active clients

Number of replicated databases

Number of replicated servers

CPU speed

Disk speed

Communication links speed

Non-scaling variables:

Collocation/distribution of software tasks

Priority scheduling

Trading algorithm

Name/directory service algorithm

Dependent variables:

Throughput: t

Response time: rt

Cost in dollars: c

Preference function:

Any system whose $\psi \geq 1$ is equally preferred, while systems with $\psi < 1$ are unacceptable.

$$\text{pref}(\psi) = \begin{cases} 1, & \text{if } \psi \geq 1 \\ \text{-10,000}, & \text{otherwise.} \end{cases}$$

This example is interesting because it represents an instantiation of the scalability framework for distributed systems in general. Such instantiation could be included on a catalog to guide analysts in the instantiation of the scalability framework for their own systems.

A second example comes from a connection management system described in the paper. The system is responsible for allocating and deallocating bandwidth, as well as setting up and tearing down the connections to previously allocated resources. The authors use the metric ψ to analyze two scalability strategies to maximize the number of active clients (with a minimum target of 30 times the number of active clients supported by the system-as-is): (A) replacing all CPUs with faster ones, and (B) replacing all CPUs with the exception of the one on which the database is executing. This objective is instantiated in the scalability framework by the performance goal Maximize[Number of active clients supported by the system]. Whatever strategy is adopted, the system must not reduce the power per invested dollar, must maximize throughput for a fixed response time, and must minimize response time for a fixed throughput. In other words, ψ must be greater than 1. These objectives are represented in the scalability framework by the goals Avoid[Reducing power per invested dollar], Maximize [Throughput for fixed response time] and Minimize [Response time for fixed throughput]. The authors consider cost directly proportional to the CPU speed. Therefore, the characteristics of the application domain and system design that are expected to affect the satisfaction of the above goals are the number of active clients, the CPU speed, and the strategies 'all machines' vs 'all but the database machine'. The scalability framework is therefore instantiated as follows:

Question:

Can the system increase the number of active clients by proportionally increasing the CPU speed of (A) all machines or (B) all but the database machine?

Quality goals:

Maximize [Number of Active Clients Supported by the System]

Maximize [Throughput for Fixed Response Time]

Minimize [Response Time for Fixed Throughput]

Avoid[Reducing Power per Invested Dollar]

Independent variables:

Scaling variables:

Number of active clients (at least 30 times the number of active clients of the system-as-is)

CPU speed

Non-scaling variables:

'All machines' vs 'All but the database machine' strategy

Dependent variables:

Throughput

Response time

Cost in dollars

Preference function:

Any system whose $\psi \geq 1$ is equally preferred, while systems with $\psi < 1$ are unacceptable.

$$\text{pref}(\psi) = \begin{cases} 1, & \text{if } \psi \geq 1 \\ \text{-10,000}, & \text{otherwise.} \end{cases}$$

By instantiating the scalability framework for the examples above, we believe to have presented information that was presented in a free-form discussion in the paper in a more concise, clear and precise form. We therefore speculate that it becomes clearer to the reader which goals and characteristics of the application domain and system design are relevant to the scalability of distributed systems and of the connection management system described in the paper.

Commercial Server-based Software

Masticola and Bondi discuss a large-scale commercial server-based software product (Masticola et al., 2005). The system, denoted by LSCSP, is partitioned into concurrent processes that communicate using socket-based transmission of SOAP messages. The authors describe an effort to develop another version of the system based on Java technology. Among the objectives of the new version were (1) to increase the number of users by one order of magnitude on a "standard server", (2) to increase the total number of users by adding more servers, and (3) to support failover between servers with minimum interruption of service. Objective (1) means that the new version would have to support a higher number of users while maintaining resource usage within the capacity of a "standard server". Objective (2) is concerned with the effectiveness of their chosen scalability strategy for achieving the satisfaction of the system goals for the expected number of users. Objective (3) should also be considered in a scalability analysis because failover operations would require cross-server request handling and data replication, which consumes resources that would be otherwise available to support user operations. These objectives are instantiated in the scalability framework by the goals Increase [Number of Users on a "Standard Server"], Increase [Total Number of Users Supported by the System], and Minimize [Interruption of Service in Case of Failover]. The authors declared that, in order to achieve these objectives, the LSCSP's modules would have to be mapped into Java containers of various types. They identified at least 10 possible process-to-container mappings. Additionally, within each mapping, several technology options were possible for communications between pair of modules. The authors list 7 communication technologies among the ones considered. The authors were also interested on the use of active resources (such as processors, bandwidth and I/O devices), on the use of passive resources (such as memory) and the on latency for inter-module communication). Finally, in addition to the scaling in the number of users, the authors were also concerned with the scaling of the message length sent between system's modules.

Despite the many aspects the authors claim to consider in their scalability analysis, the description of the problem is limited. For example, it does not describe the system's goals for the response time of user requests, or the desired latency for inter-module communication. There is also no discussion on whether the target satisfaction of these goals are allowed to vary with the scaling in the number of users and message length. The authors also did not mention any cost goal limiting the number of servers that can be added to support more users. Yet, the information available in the paper can be used to instantiate the scalability framework as follows:

Questions:

1. Can the system increase, by one order of magnitude, the number of users supported on each "standard server" by changing the implementation from C# to Java?

2. Can the system increase the total number of users through the use of collaborating servers? [12]

Quality goal:

Increase [Total Number of Users Supported by the System]

Increase [Number of Users on a "Standard Server"]

Minimize [Interruption of Service in Case of Failover]

Independent variables:

Scaling variables:

[12]The paper does not indicate the desired total number of users or any goal that will limit the number of collaborating servers.

Number of users per 'standard server' (at least 10 times the number of users on the system-as-is)

Average message length

Number of collaborating servers

Non-scaling variables:

Choice of server implementation language (C# vs. Java)

Choice of process-to-container mapping

Choice of communication technology between modules

Dependent variables:

Processors utilization

Bandwidth usage

I/O devices usage

Memory usage

Disk usage

Interruption time

Preference functions: [13] *A system design that minimizes interruption time is preferred over the others, while any system whose interruption time exceeds a given threshold is considered unacceptable.*:

$$\text{pref(interrupt\_t)} = \begin{cases} 0, & \text{if} \quad \text{interrupt\_t} > \text{max\_t} \\ \dfrac{\text{max\_t} - \text{interrupt\_t}}{\text{max\_t}}, & \text{otherwise.} \end{cases}$$

The authors use scenarios, experimentation, published study data, and performance data from baseline systems to accomplish the scalability analysis. They build a spreadsheet-based model for determining whether the non-functional requirements are likely to be simultaneously met. The authors claim that the model contains 23 different parameters. However, it is not clear from the paper which parameters are used or which non-functional requirements are expected to be met. We speculate that by describing the system goals, application domain characteristics and design alternatives according to the scalability framework, it would have been clearer to readers what the authors have accomplished.

Web-based Shopping System

Arlitt et al. present a workload analysis for a large Web-based shopping system (Arlitt et al., 2001). The system includes Web servers, application servers, database servers, load balancers, and firewall appliances. The objectives of their study are to assess the system's scalability with respect to response time for HTTP requests and to support a capacity planning exercise. For this reason, the authors are particularly interested in analyzing the system's ability to meet its goals at peak load. The authors were also interested in verifying whether simply increasing the number of application servers handling HTTP

[13]The preference function are limited to the information available in the paper. There should be other preferences (e.g., response time, message latency, costs).

requests (which they refer to as "horizontal scalability") was a cost-effective solution.

The system is expected to provide a pleasurable shopping experience to customers. This should be achieved through the provision of a personalized service to customers, and by avoiding poor site responsiveness. These objectives are conflicting, as a personalized service requires the dynamic generation of HTML resources, increasing the request response time. They are represented in the scalability framework by the goals Maintain[Personalized Service to Customers] and Achieve[Request Responded quickly]. These goals are not described in a measurable way by the authors of the paper. We, therefore, are unable to derive preference functions for this example.

Providing a personalized service requires the redirection of requests to the application server, the creation of sessions, and the dynamic allocation of resources; all of which increase the demand for CPU. CPU utilization is therefore the resource of greatest concern to the authors of the paper. Other measures of interest are peak disk utilization and physical memory utilization. These characteristics, together with response time, are represented in our framework as dependent variables.

Response time and resource usage are influenced by a number of factors: the type of customer issuing requests (users and robots), the type of request (cacheable, non-cacheable and search), the number of active sessions (containing sequence of requests from the same customer), the number of concurrent requests (from multiple sessions), the type of resources being requested (static and dynamic), the number of references per resources, and the number of application servers in the system. All these characteristics are represented in our framework as independent variables. The size of the request and the size of the response header and body were also mentioned as characteristics affecting the response time and resource usage, but they had to be left out of the scalability analysis because the authors did not have access to data to characterize them. They are, therefore, nuisance variables in the scalability framework.

We next show the instantiation of the framework for the system just described:

Questions:

1. Can the system meet its responsiveness goal for HTTP requests when the number of concurrent requests achieves its peak?

2. Is "horizontal scalability" a cost-effective strategy for supporting an increasing number of concurrent requests?

Quality goal:

Maintain[Personalized Service to Customers]

Achieve [Request Responded Quickly]

Independent variables:

Scaling variables:

Number of concurrent requests

Number of active sessions

Number of resources

Number of references to resources

Number of servers

Non-scaling variables:

Type of customer (users and robots)

Type of request (cacheable, non-cacheable and search)

Type of resources (static and dynamic)

Nuisance variables:

Size of request

Size of response headers of bodies

Dependent variables:

CPU utilization

Response time

In order to answer the above questions, the authors perform a number of workload characterizations, and analyze the impact of different classes of requests and users on the system's response time and CPU utilization. As a result of such analyses, strategies for achieving scalability with respect to response time are suggested, including caching mechanisms for dynamically generated resources and relaxing the QoS at peak times. Simply adding more servers was considered inadequate due to the significant impact of resource type and due to the workload mix in the number of server needed for meeting the system's goals.

We speculate that, in this example, the scalability framework also would have helped the authors to explain their scalability study concisely, and perhaps would have highlighted some missing information (e.g., What are the cost goals of the system? How is cost-effectiveness measured?). In fact, given the different analyses they performed for the study, separate framework instantiations would have made clear to the readers which were the relevant characteristics and metrics for each analysis.

Recast Examples from Industry Interviews

We next briefly describe scalability problems faced by real companies, and show how the elements of the scalability framework can be instantiated to describe these problems clearly and concisely. Naturally, we cannot claim that had the scalability framework been used, these problems would have been avoided. However, our framework encourages a proactive approach to scalability, where the system's goals are systematically examined for the identification and resolution of scalability risks. Given that nearly all interviewed companies had paid little or none attention to scalability requirements, they would probably have benefited from our framework.

As in the previous section, the instantiations of the scalability framework are limited to the information provided by the interviewees.

ERP System

This case reflects a company that developed bespoken ERP systems to customers in Brazil during the 90's (see "ERP System" in Appendix B). The systems were divided in Online Transaction Process (OLTP) and Online Analytical Process (OLAP) modules. The architecture was a simple SQL client that connected directly to an Oracle database. The company always started the development from the OLTP module, and normally faced problems when adding OLAP support. Furthermore, the company's customer base experienced rapid growth over the years, which led to a move from bespoken systems to a generic ERP product to be deployed at different customers. That move created many scalability problems. Attempts to scale exhausted the system resources (CPU, disk, memory, and network bandwidth) and took unacceptable response times. Resource usage and response time were influenced by a number of factors: the number of users concurrently logged into the system, the type of users (humans and external systems), the number of concurrent queries, and the type of queries (transactional and analytic). Also, developers relied on the implicit assumption that scalability could be gained by more powerful hardware. Validating such a strategy required a careful scalability analysis. Had this analysis been performed, the instantiation of the framework variables would have been as follows:

Questions:

Can the system maintain acceptable response times for a growing number of users and queries, by varying the power of its infrastructure?

Quality goal:

Maintain [Response Time for Queries]

Independent variables:

Scaling variables:

Number of concurrent users (10 to 100)

Number of concurrent queries (up to 1,000 per day)

CPU processing speed

RAM size

Disk size

Non-scaling variables:

Type of users (humans and external systems)

Type of queries (transactional and analytical)

Dependent variables:

Average/Maximum CPU utilization

Average/Maximum Disk usage

Average/Maximum Memory usage

Average/Maximum Network bandwidth usage

Average Response time

Based of this instantiation, one could examine how the growing number of users affects the satisfaction of the performance goal Maintain[Response Time for Queries]. Such an analysis would characterize the workload based on the distribution of types of users and queries, and examine whether varying the power of the infrastructure (CPU processing speed, RAM, and disk sizes) would bring the expected benefits. This analysis would probably have to take into consideration a cost goal not specified during the interview.

Field Services Systems

This example recasts a scalability problem faced by a company that specializes in building software to manage complex field service operations (see "Field Services Systems" in Appendix B). Field services systems schedule jobs for field workers, dispatch workers to specific locations and provide up-to-date and accurate information to workers.

Each system has, on average, over one hundred users (workers), each receiving five to ten jobs per day through a mobile device. Each device downloads roughly one thousand *pages* from the server at start up, and another two hundred pages during the day. In addition, the system constantly receives global positioning system (GPS) information from the mobile devices and performs automatic synchronization. All these messages consume bandwidth and may create queues waiting for processing in the system's backend server.

Despite the requirements set during the sales negotiation process, the company often faces scalability problems after the system has been put into production—usually because the amount of data (i.e. number and size of messages) received by the system exceeds the company's expectations. More data means more bandwidth usage, longer message processing times, increased queue size and longer end-user waiting times. Devices are then forced to reconnect, spending more bandwidth and incurring higher costs to the customer.

The instantiation of the framework variables reflecting this problem would be as follows:

Questions:

Can the system maintain acceptable response times and bandwidth costs for a growing number of messages and message sizes?

Quality goal:

Maintain [End-user Waiting Time]

Minimize [Monthly Bandwidth Costs]

Independent variables:

Scaling variables:

Number of concurrent users

Number of jobs per day

Number of pages at startup

Size of pages at startup

Number of pages during the day

Size of pages during the day

Number of GPS location messages

Non-scaling variables:

Size of GPS location messages

Dependent variables:

Average/Maximum network bandwidth cost

Average end-user waiting time

Average/Maximum network bandwidth usage

Average queue size

Average queue waiting time

Average message processing time

This instantiation could be used to guide an analysis of the impact of the scaling characteristics (users, jobs, pages and GPS location messages) on the satisfaction of the goals Maintain[End-user Waiting Times] and Minimize[Monthly Bandwidth Costs]. Note that, in addition to the qualities of interest (network bandwidth cost and end-user waiting times), the instantiation of the framework includes other variables (average queue size, average queue waiting time and average message processing time) that can be used to explain the values obtained for the qualities of interest.

Message Validator

This case describes a software engine that validates messages against a number of validation rules (see "Message Validator" in Appendix B). The engine is commercialized as an API, which is used by customers to build their own systems. Rules are written by customers with a graphical tool. When the system starts, rules are loaded into the engine, which then waits for messages to validate. During validation, a message is loaded entirely into memory, so it can be randomly accessed. Rules run independently, and the engine may hold a number of messages in memory. This design was chosen to satisfy one of the main goals, providing fast validation of messages.

The engine has been used to build systems for a number of different domains. The number of rules developed by customers varied greatly. The largest set of rules the system is known to be used for is 2,500. The size of the messages being validated varied between 200KB and 100MB. In addition, the complexity of the rules (i.e., the logic implemented by rules) and the complexity of the message (i.e., its structure) also varied. All these factors influence the message validation time.

Recently, the owner of the company has been approached by a customer interested in validating documents as big as 2GB. The relationship between validation time and document size have been investigated by the company. The system's inability to validate large messages is a known scalability issue, because the machine memory becomes exhausted.

The instantiation of the framework reflecting the scalability concern just described is as follows:

Questions:

Can the system achieve acceptable message validation times for a growing message size?

Quality goal:

Achieve[Message Validated Quickly]

Independent variables:

Scaling variables:

Number of messages per second

Size of messages

Complexity of messages

Number of validation rules

Complexity of the rules

Dependent variables:

Message validation time

Maximum memory usage

Nevertheless, not all aspects represented in the instantiation of the framework above have been considered in the company's analysis of the system's scalability (e.g., complexity of message and rules). These aspects only appeared during the interview, as we attempted to identify the factors that could influence the goal Achieve[Message Validated Quickly].

7.6.2 Examples of the Elaboration of Scalability Requirements

We now illustrate the identification of scalability obstacles and possible resolutions for a well known and widely studied real-world example in the requirements engineering literature, the London Ambulance Service (LAS) system. Other real-world examples of scalability obstacles and their resolution are described next.

The London Ambulance Service (LAS)

The computer-aided system (CAD) develop for LAS has two main functions: responding to emergency calls requiring the rapid intervention of an ambulance, and dealing with non-urgent patient journeys. Its responsibilities include receiving calls, dispatching ambulances based on an understanding of the nature of the call and the availability of resources, and monitoring the progress of the response to the call (Finkelstein and Dowell, 1996). This example is primarily based on the *Report of the Inquiry Into the London Ambulance Service* (Page et al., 1993) and goal models published in previous works (Letier, 2001; van Lamsweerde and Letier, 2000).

Description of the System

When a 999 or urgent call is received in the Central Ambulance Control (CAC) room, the Control Assistant (CA) fills in a form with the details of the incident, including the coordinates of the incident location. These goals are captured by the portion of the goal model in Figure 7.23. The goal Achieve[Incident Reported] is refined into two subgoals. The goal Achieve[Emergency Call Taken] describes the goal of answering an emergency call made by a member of the Public. Calls are taken by Call Assistants with the support of a Call Handling Software. The goal Achieve[Accurate Incident Call] is satisfied by accurately identifying the incident location through a Gazetteer and Map Software and encoding this location into the incident form, a responsibility of the Call Assistant. These subgoals are represented in Figure 7.23 by the nodes

Achieve[Accurate Incident Location] and Achieve[Incident Form Encoded with Incident Location], respectively.

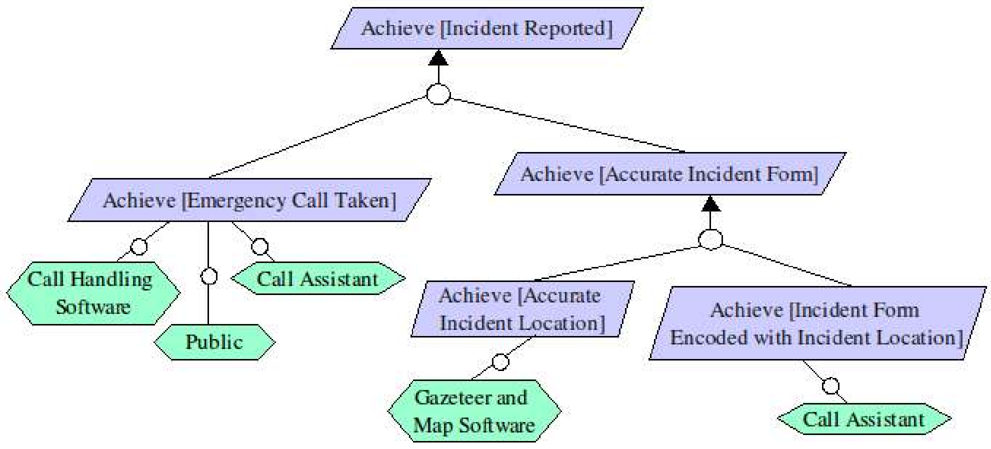

Achieve [Incident Reported]
Achieve [Emergency Call Taken]
Achieve [Accurate Incident Form]
Call Handling Software
Call Assistant
Public
Achieve [Accurate Incident Location]
Achieve [Incident Form Encoded with Incident Location]
Gazeteer and Map Software
Call Assistant

Figure 7.23: LAS incident reported. Adapted from van Lamsweerde and Letier (2000).

Once the form is encoded, available ambulances are allocated optimally according to the incident location. The allocated ambulance is then mobilized by informing the relevant ambulance station or the ambulance directly, if it is away from the station. Mobilizing an ambulance at the station (Achieve [Allocated Ambulance Mobilized at Station]) requires an order to be printed (Achieve [Mobilization Order Printed at Station]) and an ambulance to be mobilized from that order (Achieve [Ambulance Mobilized from Printed Mobilization Order]). Printing orders involves sending them to the station printer (Achieve [Mobilization Order Transmitted to Station Printer]) and printing them (Achieve [Received Mobilization Order Printed]). These goals are shown in Figure 7.24. Mobilizing an ambulance on the road requires displaying a mobilization order on the chosen ambulance's mobile data terminal (Achieve [Mobilization Order Displayed on MDT]) and then mobilizing the ambulance according to this order (Achieve [Ambulance Mobilized from Order on MDT]). The former goal is achieved by identifying the mobile data terminal in the allocated ambulance (Achieve [Accurate Ambulance/MDT Mapping]), sending it a mobilization message (Achieve [Mobilization Order Sent to Mapped MDT sent to the MDT]), ensuring the message is transmitted (Achieve [Mobilization Order Transmitted to the MDT]), and displaying the message on the ambulance's mobile data terminal (Achieve [Received Allocation Displayed on MDT]). These goals are illustrated by Figure 7.25.

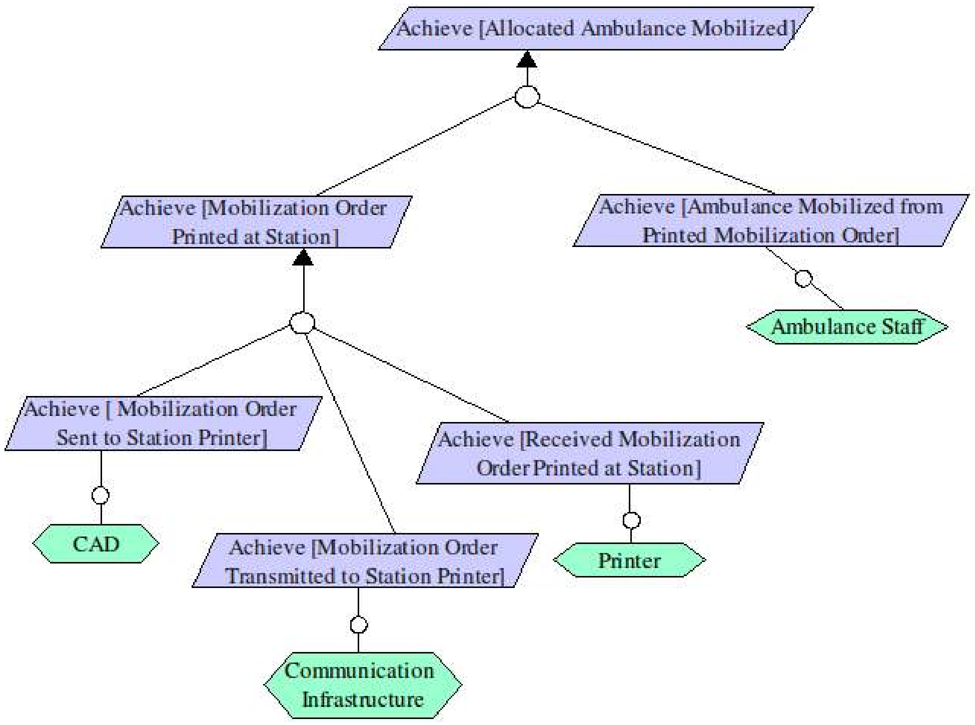

Achieve [Allocated Ambulance Mobilized]
Achieve [Mobilization Order Printed at Station]
Achieve [Ambulance Mobilized from Printed Mobilization Order]
Ambulance Staff
Achieve [Mobilization Order Sent to Station Printer]
Achieve [Received Mobilization Order Printed at Station]
CAD
Achieve [Mobilization Order Transmitted to Station Printer]
Printer
Communication Infrastructure

Figure 7.24: LAS ambulance allocated at station. Adapted from Letier (2001).

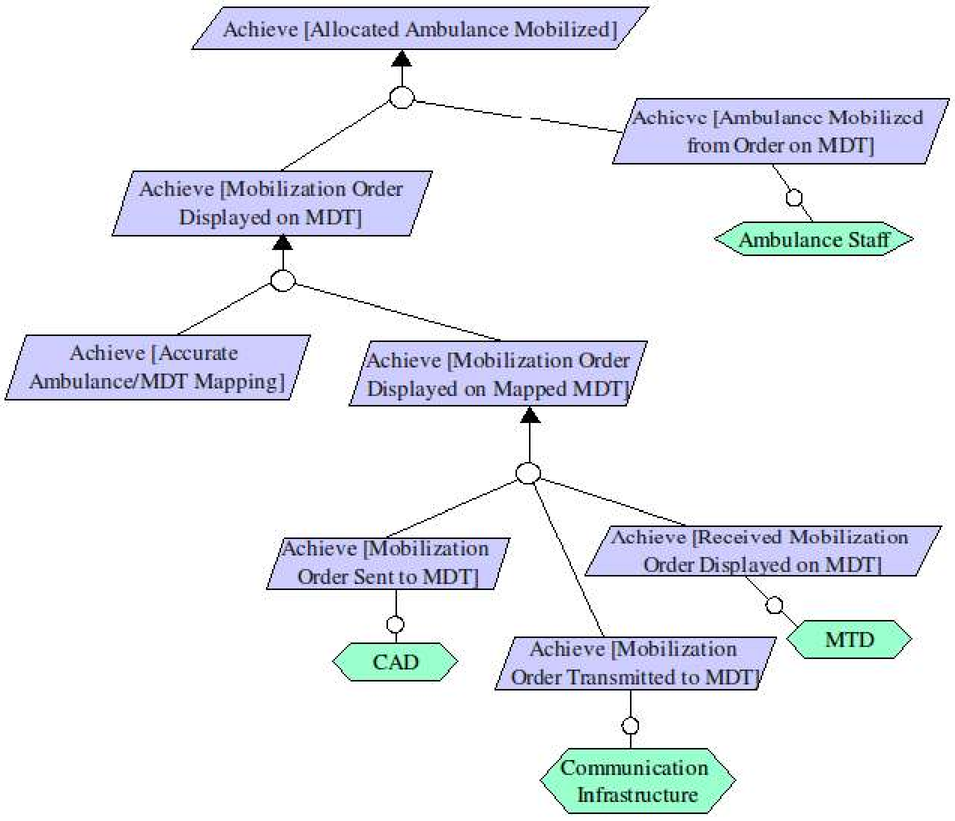

Achieve [Allocated Ambulance Mobilized]
Achieve [Ambulance Mobilized from Order on MDT]
Achieve [Mobilization Order Displayed on MDT]
Ambulance Staff
Achieve [Accurate Ambulance/MDT Mapping]
Achieve [Mobilization Order Displayed on Mapped MDT]
Achieve [Mobilization Order Sent to MDT]
Achieve [Received Mobilization Order Displayed on MDT]
CAD
Achieve [Mobilization Order Transmitted to MDT]
MTD
Communication Infrastructure

Figure 7.25: LAS ambulance allocated at road. Adapted from Letier (2001).

Obstacle Identification

As modeled, the goals above do not refer to scaling assumptions on domain quantities. Their load is therefore unbounded, giving rise to a number of scalability obstacles:

Number of Calls Exceeds Call Assistant Team Capacity: This obstacle may prevent the call assistant team to take an emergency call in time, and is caused by a larger volume of emergency calls than the call assistant team can handle, leading to unacceptable ringing times.

Number of Calls Exceeds Call Handling Software Capacity: This obstacle may lead to an ongoing call to be dropped or a new one not getting through because the volume of emergency calls is larger than the call handling software can support.

Number of Location Requests Exceeds Gazetteer and Map Software Capacity: This obstacle may prevent the accurate incident location in time, and is caused by a volume of location requests that is larger than the gazetteer and map software capacity.

Number of Messages Exceeds Communication Infrastructure Capacity: This obstacle is caused by an overload of the communication infrastructure capacity, and may prevent messages from being delivered to MDTs and stations' printers.

Number of Printing Requests Exceeds Printer Capacity: This obstacle may obstruct reports from being printed at stations due to overflow of the printer queue, insufficient paper or used-up cartridges.

Message Size Exceeds MDT Display Capacity: This obstacle may obstruct mobilization orders from being shown because the amount of information to be displayed is greater than the MDT can handle.

None of these obstacles have been identified in a previously published list of obstacle refinements for the LAS system (Letier, 2001).

Obstacle Assessment

Once the obstacles have been identified, their criticality and likelihood must be assessed. Our illustrative example of LAS is based on a few key documents describing the system (Letier, 2001; Finkelstein and Dowell, 1996). Since we did not have access to the stakeholder or sufficient information about the problem domain, we limit this section to exemplify the kinds of considerations to be taken when assessing two of the identified scalability obstacles:

Number of Calls Exceeds Call Assistant Team Capacity: In order to assess the likelihood of this obstacle, one must understand the nature and volume of incoming calls, as well as the load they impose on control assistants. For example, do all calls demand the same time from the control assistant? What is the average time required by each call category? What is the distribution of calls among these categories? How is the number of calls expected to vary during the day, over the year, and in the next few years? Which scenarios may lead to an increase in the number of calls? How likely are they?

For example, the requirements analyst may discover that callbacks require more time than new calls. He may also find that heavy traffic or inefficient ambulance allocations lead to delays in ambulances and, consequently, to an increased number of callbacks. In these cases, more questions should be asked: What is the likelihood of heavy traffic? What may cause inefficient ambulance allocations?

One should also ask what would happen if emergency calls are not answered within the specified ringing times. In other words, how critical is that to the system?

Number of Messages Exceeds Communication Infrastructure Capacity: Similarly, in order to understand the likelihood of this obstacle, one should investigate the size and volume of different types of messages being transported by the communication infrastructure. For example, when are messages sent? Where are messages sent to? Are there different types of messages? What is the size of the different types of messages? Do messages need to be acknowledged? Are messages or acknowledgments ever resent? Which scenarios may lead to an increase in the number of messages on the network? How likely are they?

In the LAS, messages may be sent to stations or directly to ambulances, and can be of different types, such as allocation, mobilization, login and acknowledgement messages. Retrospectively, we know that the number of messages increased due to software delays and mistrust in the system from the LAS staff (Page et al., 1993). Perhaps these scenarios could have been uncovered by a careful cause-effect analysis of scalability obstacles.

As in the previous example, one should also ask what happens if messages get lost or delayed? Is the delivery of every type of message equally critical?

Obstacle Resolution

As none of the goals in the original model referred to assumptions on domain quantities, the first step is to apply the resolution tactic *introduce scaling assumption*. The following is a simplified example of a scaling domain assumption on the number of incoming calls:

Assumption 7.2

Assumption Expected Number of Incoming Calls
Category Scalability
Definition The total number of calls includes ordinary and emergency (999) calls. On average, the LAS receives between 2,000 and 2,500 calls daily; this includes between 1,300 and 1,600 emergency calls.[a]

[a] Numbers are taken from Finkelstein and Dowell (1996). In an actual requirements engineering exercise, this assumption should include a careful characterization of the incoming calls, including the different categories of calls, their distribution, and estimation over the following years, and so forth.

We next illustrate further possible resolutions to these obstacles:

Number of Calls Exceeds Call Assistant Team Capacity: A number of tactics can be applied to resolve this obstacle. One can, for example, *adapt agent capacity at runtime according to load* by increasing the size of the call assistants team on duty to support the scaling in the number of calls. It is also possible to *split goal load by case*, where new calls and callbacks are dealt with by different teams of call assistants.

Number of Messages Exceeds Communication Infrastructure Capacity: Mitigating this obstacle may be a matter of ensuring the deployment of a communication infrastructure with sufficient bandwidth to support the anticipated number of messages being communicated, or in other words, *set agent capacity upfront to worst-case load*. Alternatively, one can *adapt agent capacity at runtime according to load*, where the message volume is monitored and the system administrator is informed of a growth trend in the number of messages, so that the infrastructure can be upgraded to prevent the scalability obstacle.

Scalability Obstacles and Resolutions in Industry Interviews

This section exemplifies real-world scalability obstacles and resolutions observed during the industry interviews. Given the very limited information we had about these systems, our intend is not to identify scalability risks or resolutions other than the ones described by the interviewees themselves. We aim instead to demonstrate how scalability problems faced by real-world companies in different application domains, and their resolutions, can be expressed at the goal level, by using the concepts and tactics described in Chapter 5.

Data Integration System (DIS)

This system aggregates data from various external data sources and makes this data available to a number of internal applications in a large transportation company. The system has a number of well-defined non-functional requirements, including a goal on the response time for data requests by internal applications. We refer to this performance goal as Achieve [Data Provided Quickly for Expected Number of Application Requests]. The system also has other maintenance goals such as Achieve [Historical Data Updated from Recent Data], Achieve [Recent Data Gathered from External Data Source], and Achieve [Obsolete Data Removed]. These maintenance goals consume a lot of the machine's resources (in particular CPU cycles). All these goals may be obstructed by the obstacle Number of Requests Exceeds DIS Capacity, where the number of requests is the sum of application requests and maintenance requests that satisfy the goals above. In order to mitigate this obstacle, the system designers defined maintenance time frames and scheduled these heavy tasks to run in these time frames. This resolution corresponds to the tactic *limit goal load according to fixed agent capacity*.

However, despite the application of this tactic, sometimes maintenance tasks overlap. This situation increases the number of requests the DIS has to handle, obstructing the satisfaction of the performance and scalability goal Achieve [Data Provided Quickly for Expected Number of Application Requests]. The overlapping of maintenance tasks may be caused by a number of reasons, such as (1) an external data source becoming unavailable, forcing the system to wait and delaying the execution of a maintenance task; (2) a system backup running without previous agreement, overloading the CPU and causing a maintenance task to take longer then expected; or (3) an external system sending more requests than expected and, again, causing maintenance tasks to run for longer.

The company's solution was to accept that overlapping would sometimes occur, and to have developers constantly on call to stop the system, find a time where the CPU would be less busy, and manually reschedule maintenance tasks. This resolution belongs to the category *goal restoration*.

Communication Services

This example describes a company that develops software for telephone operators, telecom operators, and cable companies. They offer a set of services that vary from dial-up ISP authentication, to e-mail provisioning, to digital television (see "Communication Services" in Appendix B). Their first customer had one million users and was planning to grow by ten thousand users a week. This expectation would be represented in a goal model as a scalability assumption stating that the expected number of users would vary from one to three million in the first three years. System goals such as Achieve [Authentication of Dial up User Quickly] would then have to be satisfied for the expected number of users. This increase in the number of users threatened the satisfaction of a number of system goals. This risk could be represented in the goal model as the scalability obstacle Number of Users Exceeds the System Capacity.

To mitigate this risk, the system was built with scalability in mind from the outset. The first scaling strategy adopted by the company was to replicate the system as a monolithic instance on multiple servers to spread the load when the machine was reaching its limits. This strategy corresponds to the tactic *adapt agent capacity at runtime according to load*. The company later realized this solution was wasteful, because the system had two types of end user with very different usage profiles, with one requiring a much faster service than the other. Replicating a monolithic system did not solve the problem for users requiring a faster service. The company then decided to break the system down into services that could scale individually, spreading the load according to the type of incoming request. This solution corresponds to the tactic *split goal load by case*.

Funding Compliance System

Web Application

This example refers to a family of Web-based products for telecommunication operators (see "Web Application" in Appendix B). The company specializes in novelty products, whose popularity is difficult to predict. In the same way that a new product can be a sudden success, its use may drastically decrease when a successful promotion gets to an end. For this reason, instead of trying to anticipate the popularity of a new product, the company adopts a scaling strategy to mitigate the scalability obstacle Number of Requests Exceeds System Capacity.

Products are built as a number of Web servers running on commodity machines to which the load is dynamically distributed. The strategy is to scale horizontally, adding more hardware as the demand for a particular application increases. This strategy is built into the company's business model. New applications are normally deployed in shared servers, and may receive a dedicated server if they prove to be a success. Load analysis is performed weekly, sometimes daily, and the number of servers is adjusted according to the demand. They also maintain a few idle machines to support an increase in the load due to a marketing campaign or some other reason. This resolution illustrates the use of the tactic *adapt agent capacity at runtime according to load*.

Telephone Billing System

This system is responsible for the billing of telephone calls (see "Telephone Billing System" in Appendix B). A call is composed of a number of messages that contain a series of information, such as the caller and callee identities, the duration of the call, and who terminated the call. The system captures these messages from the telecom operator's network, identifying each message, and reconstructing the call before it is saved on the database to be made available to a number of business applications.

In order to achieve the goal Achieve [Calls Reconstructed], the system must, among other things, Avoid [Message Loss]. This subgoal should be achieved for the scaling assumption Expected Number of Messages. The system designers decided to satisfy this goal by keeping, in a memory buffer, a copy of each message passed to the system's network until the call has been correctly reconstructed and saved.

Although the system has never halted due to scalability problems, it has produced inconsistent results due to an overflow of the memory buffer and, consequently, the loss of messages. In other words, the scalability obstacle Number of Messages Exceeds System Capacity obstructed the satisfaction of the goal Avoid [Message Loss]. The reason for the increase in the number of messages was long network downtimes.

The company's solution was to migrate the system to a 64-bit machine. The new machine had a much larger address space, which prevented the overflow of the memory buffer. This resolution corresponds to the tactic *set agent capacity upfront to worst-case load*.

7.6.3 An Informal Study: The "Realtime Banking System"

An informal study was executed to aid the design of a comparative case study to investigate how the scalability framework could be applied during the software development lifecycle. The actual study was a coding task to be given to undergraduate students as a coursework. However, in order to certify that the case study would run smoothly, the coding task was first given to two professional developers. The developers were asked to implement a simple concurrent system, following a set of systematic steps. The steps themselves had not yet been formalized, and the study was run in a very informal way. The results were: (1) feedback from professional developers on the scalability framework and a planned case study, and (2) a first sketch of a method for scalability analysis.

Study Description

This study consisted of implementing a simple real-time banking system. The system was required to verify requests for money transfers, ensuring that the payer has enough money in their account to perform the transfer, and that the transaction is unlikely to be fraudulent. When a money transfer request is received, the system performs the following tasks: (1) it checks the balance in the payer account, (2) it checks the request against a set of pre-defined rules to assess if the transfer is suspicious, (3) it performs the transfer, updating the balances of payer and payee; and (4) it notifies the client of the completion or denial of the transaction. In order to check the payer balance and the likelihood of fraud, the server maintains, for each account, information including the origin country of the account and a summary of all transactions in the current month. This information is used by fraud analysis rules to decide if the transfer request is suspicious.

Developers were given a hypothetical distribution for request arrival rates and the required throughput, and were told that a bank is expected to deal with over 1.5 million accounts. Developers were also requested to implement the system using only standard Java classes, without a database. The maximum allowed heap size was 64MB, and disk was 100MB. Because of the lower heap size, developers could not keep all accounts information in memory, having to extend storage to disk, while maintaining throughput.

Summary of the Results

Both developers, one with four and the other with fifteen years experience in software development, attempted to implement the system by following our instructions. Due to time constraints, only one delivered a working system, with rough measurements of dependent variables. Although the study was not conducted rigorously, it resulted in a first sketch of a method for scalability analysis.

Both developers had a very basic understanding of scalability prior to the study, mostly relating it to performance. The developers have stated that they agree with the majority of the points made by the research and admitted that they would not have thought about scalability if they had not been requested to do so. This is not surprising, especially because neither of them had a lot of experience with architecting systems. They found that the framework let to good practice, but also demonstrated concerns with the overhead of performing a scalability analysis. Both developers said they have gained a greater understanding of scalability and that the experience has influenced the way they will develop systems in the future.

Critical Evaluation of Informal study

As a one-off informal study, very limited conclusions can be drawn. Yet, the study was useful in the design of a case study and as a starting point to develop a scalability analysis method. The framework does seem to have changed the volunteers' perception of scalability and to be considered useful in software development. We have also learned that there was a concern with the overhead of the analysis and certain resistance in following it, step-by-step, in the real-world.

7.7 Critical Evaluation

This thesis concentrates on a definition for scalability, a technique for elaborating scalability requirements, and an analysis technique for quantitatively characterizing and comparing the scalability of software systems. The thesis also describes a method that brings together these concepts and techniques.

Despite the effort made to validate these techniques against real-world systems, data and stakeholders, the case studies did not cover the whole method described in Chapter 6, but rather only parts of it. In none of the studies was the method followed from the beginning to the end—i.e, define scalability question, build preliminary model, update goal model, select goals for scalability analysis, refine and define new scalability questions, instantiate variables and functions, prioritize goals, perform analysis, and state scalability answer or claim. In particular:

1. Case study 3 illustrated the instantiation of variables and functions for two specific scalability analyses, and considered only portions of the goal model. These simple analyses did not require the combination of distinct scalability objectives, and consequently no utility function was needed. Therefore, initial ideas on using requirements prioritization techniques for deriving utility functions from a goal model have not been validated.

2. Case study 3 coincided with the construction of the third version of the IEF. By the end of the study, the system development was still in the early stages of the design phase. On one hand, that allowed us to take into consideration some system design characteristics in the instantiation of variables for the scalability analysis. On the other hand, it was not yet possible to validate whether the instantiated scalability analysis would help to predict the actual system scalability accurately. Furthermore, the selection of variables representing system design characteristics and the conversion from objective functions in the goal model to preferences were performed in an ad-hoc manner. It is, therefore, possible that not all relevant dependent and independent variables from the system design have been taken into account, or that the resulting preferences precisely translate the stakeholders' scalability goals.

3. Case studies 1 and 2 showed that it is possible to characterize scalability quantitatively, while case study 3 demonstrated that one can derive from a goal model the variables and functions for a scalability analysis. None of the studies, however, demonstrated that the proposed techniques indeed result in the construction of more scalable systems. Doing so requires a comparative empirical study such as the case study planned in Appendix C, but not executed due to reasons beyond our control.

Combined Contribution of Case Studies and Examples

The individual benefits and limitations of each case study have been discussed in its respective section. We next assess the combined contributions of the case studies and examples to the validation of the ideas presented in this thesis. For such, we first review the research hypothesis:

The Research Hypothesis *It is possible to (1) provide a precise definition of scalability that is applicable to any application domain and (2) develop systematic techniques, also applicable to any application domain, to characterize and analyze the scalability of software systems, so that developers can identify and avoid scalability problems that would otherwise be overlooked without the definition and techniques.*

Ideally, the validation of the research hypothesis would have been done by means of a single, comprehensive case study. However, both due to the way the research has developed and time constraints of the PhD candidate and the other people involved in the studies, we have decided to perform a series of complementary studies. The first two case studies validated the scalability analysis technique, while the third one validated the techniques for elaborating scalability requirements. In addition, we have described a number of smaller examples for both techniques in order to demonstrate their applicability to different application domains and system designs.

In a retrospective study, such as case studies 1 and 2, it is difficult to claim the analysis technique would have predicted a scalability problem developers did not. We have, therefore, opted for demonstrating that the analysis technique can be used to formulate qualitative judgment about the system's scalability that is consistent with the judgments an expert on that system would make. In case study 3, we opted for showing that one can derive, from a goal model, the variables and functions that would otherwise be overlooked without the technique. More precisely, this chapter has shown that:

1. It is possible to characterize the scalability of a real-world system clearly and objectively in terms of independent variables, dependent variables, preference functions, and utility functions; and to form qualitative judgments about the system's scalability that is consistent with the judgment an expert on that system would make.

2. One can identify scalability goals, scaling assumptions, scalability obstacles, and scalability resolution strategies for a real-world system; and derive, from a goal model, variables and functions that would otherwise be overlooked.

3. One can apply both techniques to real-world systems of different application domains and system designs.

The combination of these results is a reasonable indicator of what would have happened if a single comprehensive study had been performed. That is, taking into consideration that the requirements engineering technique allows one to identify variables and functions that would otherwise be overlooked, and that the analysis technique can use these variables and functions to produce a fair qualitative judgment about a system's scalability, then the combination of both would allow one to identify and avoid problems that would otherwise be overlooked without the techniques.

Contributions and Benefits

The contributions associated with this chapter are as follows:

Contribution: *Significant case studies demonstrating the applicability of the scalability definition, requirements engineering technique, and analysis technique.*

The techniques described in this thesis were validated against a complex, proprietary, real-world system. The PhD candidate was placed within a real company, with full access to source code and documentation, and counted on the cooperation of real stakeholders. The report of such an experience is, in itself, a valid contribution to software engineering research. The PhD candidate has faced a number of issues commonly found in an industrial setting, such as the lack of time of the stakeholders, incomplete and inconsistent documentation, missing data, disagreements between stakeholders and their unwillingness to take responsibility for estimates. These difficulties led to improvements to the techniques and highlighted a number of limitations that could have passed unnoticed had the case study not been performed within an industrial setting.

An additional benefit of this chapter is

Benefit: *Recast examples from the computing literature and industry in terms of the scalability framework.*

In addition to reviewing the state-of-the-art in scalability research, we have conducted a number of interviews with stakeholders and professional developers about scalability problems they faced throughout their careers. We have used cases described in both the computing literature and in these industry interviews to demonstrate parts of the scalability framework. These smaller examples were limited by the amount of information we had access to, but served to show the applicability of the scalability framework to different application domains and system designs.

7.8 Summary

This chapter described case studies and examples used to validate the concepts and techniques presented in this thesis. These were

1. A small-scale empirical study that compared the scalability of two prototypes representing consecutive IEF versions. The study provided an early validation of the analysis technique and gave feedback for the research.

2. An empirical study that repeated the previous study for the real implementation of both IEF versions. This study included a more careful characterization of the application domain and confirmed what was known as fact about the system's scalability. The study has also highlighted limitations of the technique, which motivated the extensions to KAOS for elaborating scalability requirements.

3. The goal-oriented analysis of a new IEF version, demonstrating the use of scalability goals, scaling assumptions, scalability obstacles, and scalability resolution strategies. This study has led to the improvement of our extensions to KAOS and highlighted a number of possible paths for future research.

4. An informal study with two volunteers to investigate how the scalability framework could be applied during the software development lifecycle. The study, although performed informally, has helped to define the method in Chapter 6.

5. Examples from the computing literature and from industry interviews recast according to the scalability framework, which included the Google search engine, a distributed heterogeneous service system, a commercial server-based system, a Web-based shopping system, an ERP system, a field services system, and a message validator system. These examples demonstrated how scalability concerns from different application domains can be characterized in terms of independent and dependent variables and preference functions.

6. Examples from the computing literature and from industry interviews illustrating the identification, assessment and resolution of scalability obstacles, which included the London Ambulance Service (LAS) system, a data integration system, a communication services system, a funding compliance system, a Web application, and a telephone billing system.

Chapter 8

Conclusions

This chapter concludes our journey through the topic of software systems scalability. It recapitulates the research problem and goal, lists the main contributions, provides a critical evaluation of the work, and, finally, discusses possible directions of future work.

8.1 Review of Research Problem and Goal

Scalability is a term that appears frequently in computing literature, but it is a term that is poorly defined and poorly understood. It is an important attribute of computer systems that is frequently asserted but rarely validated in any meaningful, systematic way. The lack of a uniform, consistent and systematic treatment of scalability makes it difficult to identify and avoid scalability problems, to describe the scalability of software systems clearly and objectively, to evaluate claims of scalability and to compare claims from different sources.

Therefore, the research problem of this thesis has been stated as follows: *The computing community is lacking (1) a precise definition of the term scalability and (2) a systematic, uniform and consistent treatment of scalability that can be applied across application domains and system designs.* Proving that a technique can be applied across different application domains and system designs would be challenging in the time frame of a PhD research programme. For this reason, our aim was to develop a definition and techniques that use no application domain-specific terms and make no assumptions about the system design, but are sufficiently powerful to identify and avoid scalability problems that would otherwise be overlooked.

8.2 Thesis Contributions

The contributions of this thesis to software engineering are summarized below:

Contribution: *A uniform and precise definition of scalability that is independent of the application domain and system design.*

We define scalability as the ability of a system to satisfy its quality goals to levels that are acceptable to its stakeholders when characteristics of the application domain and system design *vary* over expected ranges.

Unlike other definitions in the computing literature that relate scalability with particular metrics, such as throughput or response time, scaling characteristics, such as number of requests or processors, or scaling behavior, such as linear or sub-linear, our definition describes scalability in terms of quality goals and scaling characteristics of the application domain and the system design. These are concepts common to all software systems, resulting in a uniform and precise definition that transcends application domains, system designs and specific system concerns.

Furthermore, by demonstrating the correctness and usefulness of such a definition we have explicitly shown the relationship of scalability with other system qualities and presented counter-arguments to common misconceptions about scalability, such as the belief that scalability is always related to performance or that scalable systems should present an at-most-linear increase in resource usage as the demands on the system increase.

Contribution: *A framework and a method for characterizing and analyzing software systems scalability that is independent of the application domain and system design.*

We have created a framework (Figure 4.1) to reveal—in a precise and explicit form—the impact of scaling characteristics of the application domain and system design on the satisfaction of the system's quality goals. The framework combines techniques for requirements engineering and scalability analysis, ensuring that the variables and functions used for the analysis are relevant to the problem at hand.

Our method brings together these techniques, describing the steps required to instantiate the elements of the scalability framework. Starting from an initial scalability question, the analyst updates a goal model through the identification, assessment and resolution of scalability obstacles. Resulting gals are then selected and used to refine the scalability question, and to derive variables and functions to be used in the analysis that will produce an answer to the scalability question. This method can be applied at any point of the development lifecycle. When applied earlier, it encourages a proactive approach to scalability when building systems.

The scalability framework follows the general steps defined by EPA and relies on concepts that are common to software systems in general, such as agents, quality goals and software metrics. It, therefore, applies to software systems regardless of the application domain and the system design.

Contribution: *A technique for describing, modeling and analyzing scalability requirements, and the description at the goal level of common strategies to resolve scalability obstacles identified during requirements engineering.*

We have developed techniques for describing, modeling, and reasoning about scalability requirements. The approach consists of systematically identifying scalability obstacles to the satisfaction of goals; assessing the likelihood and severity of these obstacles; and generating resolutions to them. The result is a consolidated set of requirements in which important scalability issues are anticipated through the precise, quantified specification of scaling assumptions and scalability goals.

These techniques bring a number of benefits to the elaboration of scalability requirements, such as uncovering and specifying non-obvious time-varying assumptions on the application domain, providing rationale for scalability goals, providing traceability for changing assumptions on application domain quantities, assigning responsibilities for satisfying scalability goals, and providing measurable quality variables and objective functions to be used for scalability analysis. Goal-obstacle analysis has shown to be a systematic and intuitive way to reason about scalability during the early stages of software development. It generates a comprehensive set of scalability obstacles and provide an accurate model of impact for these obstacles. The technique is particularly useful for recognizing scalability problems caused by exceptional circumstances and unexpected system usages.

We have also described strategies for achieving scalability goals at the requirements level, when more freedom is available for exploring alternative system proposals. As an additional benefit, the resulting goal model can serve as input to various methods for analyzing software scalability at the architectural level, and can have different uses during the development lifecycle.

Contribution: *An analysis technique for evaluating and comparing scalability characteristics of software systems that is independent of the application domain and system designs.*

The scalability analysis technique described in this thesis explains the effects that the scaling of application domain and system design characteristics (independent variables) have on the satisfaction of quality goals (dependent variables and stakeholders' preferences and utility). Consequently, any system analysis—performance, reliability, availability or other—conducted with respect to a variation over a range of application domain or system design characteristics can be modeled as a scalability analysis.

These variables and functions allow one to describe, with clarity and precision, whether the system meets its quality goals when characteristics of its application domain and system design scale, as well as to show the system's limits and scaling trends. That is, the analysis technique creates a precise and uniform vocabulary that stakeholders can use to articulate scalability claims.

Contribution: *Significant case studies demonstrating the applicability of our scalability definition, framework, requirements engineering technique, and analysis technique.*

Three case studies demonstrated the applicability of the techniques described in this thesis. The first two studies validated the scalability analysis technique, while the third one validated the technique for elaborating scalability requirements. Validation was performed against a complex, real-world system, to which the PhD candidate had full access and for which the candidate was able to enjoy the cooperation of real stakeholders. The report of such experience is in itself an important contribution to software engineering, demonstrating the benefits and challenges of applying the proposed techniques in an industrial setting.

Additional benefits brought by the validation of these techniques were their application to smaller examples taken from the computing literature and software industry, and a number of interviews with stakeholders and professional developers describing real scalability problems, with reflections on the development mistakes and lessons learned.

8.3 Critical Evaluation

This thesis aims to provide a precise definition of scalability and develop systematic techniques to characterize and analyze the scalability of software systems, as stated by the research hypothesis in Chapter 1.

Scalability characterization and analysis is a large research area, which includes workload characterization, requirements elicitation, system modeling and simulation, test design, and so on. This work concentrates on a definition for scalability, a technique for elaborating scalability requirements, and an analysis technique for quantitatively characterizing and comparing the scalability of software systems. The research hypothesis was validated through a mix of argumentation and experimentation. In particular, the usefulness and correctness of our definition for scalability have been argued through discussions in Chapters 1 to 3, while the techniques have been validated through case studies and other smaller examples taken from the computing literature and from industry interviews, as described in Chapter 7.

Despite the constrained scope of the research, the chosen areas of investigation are themselves wide and complex, and, for this reason, have not been completely addressed by the techniques described in this thesis. We next describe such limitations:

Limitations of the Scalability Analysis Technique

An analysis that relies on metrics and optimization functions imposes a few risks. If variables and functions have not been adequately selected and the study has not been carefully designed and executed, the analysis results may be misleading.

Taken in isolation, the scalability analysis technique is described in terms of generic elements. We have opted for not providing a catalog of possible instantiations of these elements for particular types of systems or scalability concerns. Thus, critics of the analysis technique may argue that the usefulness of its results may be jeopardized by the wrong choice of variables and functions. This is a real risk, especially considering that scalability problems are often caused by characteristics of the application domain and system design that have been overlooked during the construction of the system, and that the operational distribution of application domain characteristics may not be known in the early stages of the development lifecycle. This risk is unavoidable. However, in order to reduce its likelihood we complement the scalability analysis with the requirements elaboration technique described in Chapter 5.

It is also the case that one has to trust that the metrics being collected are a good representation of the system qualities they represent. Furthermore, metrics are very sensitive to the way they are collected, particularly to small errors in measurements, abnormalities and wrong assumptions. This is a problem with methods that use metrics in general. However, as argued by Fenton and Pfleeger, the subjectiveness normally associated with metrics, although a drawback, does not prevent us from gathering useful information about the system (Fenton and Pfleeger, 1996).

The technique is also limited in the sense that it does not incorporate uncertainty into the analysis or allow for the extrapolation of its results. For the purpose of this research, it is assumed that no uncertainty exists in relation to the distribution and scaling of independent variables, and that there is a running version of the system and sufficient data and hardware infrastructure to run this version. However, these assumptions will rarely hold in the real-world. The importance of incorporating uncertainty and extrapolating the analysis results is undeniable, and both topics have been set as future work.

Finally, although theoretically, raw data can be produced for the scalability analysis through modeling or simulation, this belief has not been validated by the case studies in the research. We intend to, in the future, investigate the integration of the scalability framework with existing modeling techniques, such as queuing networks and Petri Nets for performance modeling and reliability growth models (Lyu, 1996). The case studies have also not contemplated system qualities other than performance-related ones. We had planned to perform a scalability analysis of the Condor system (University of Wiscousin-Madison, 2006) with respect to reliability, but due to time constraints this study could not be performed and is also left for future work.

Limitations of the Requirements Elaboration Techniques

As mentioned above, this research assumes there is no uncertainty in relation to the values of the application domain quantities (independent variables). However, in the validation of our extensions to KAOS against a real-world system, we observed that stakeholders had different views on the values assumed by domain quantities. The many uncertainties about their future values made stakeholders unwilling to provide estimates. Reaching agreement on the projected values for domain quantities in the scaling assumptions was our most significant difficulty. As it stands, our technique for elaborating scalability requirements gives no support to model disagreements about the content of a scaling assumption, nor to model or reason about uncertainties concerning these assumptions.

Another limitation of our extensions to KAOS is that only a lightweight assessment of the load imposed on agents has been considered. However, in reality, a number of considerations must be made. First, not all goals will mobilize the agent's resources at the same time. In addition, some of the load imposed on the agent may be determined by the system design (e.g., the storage of rule results in the IEF), which is not captured by the goal model. The likelihood of a scalability obstacle affecting an agent is another issue, as it should involve identifying how the obstacle's likelihood varies with the agent's required capacity. Further techniques are needed to perform a precise quantitative assessment of the load imposed on an agent.

The requirements engineering techniques described in this thesis also do not offer support for the selection of the preferred resolution strategy from among a set of alternatives. During the requirements engineering process, a number of resolutions are likely to be generated. Choosing from among them requires being able to perform a quantitative analysis of the likelihood of scalability obstacles and their impact on goal satisfaction, as well as perform a cost/benefit analysis of each strategy.

Finally, there is a risk that a scalability goal-obstacle analysis as a pre-step for the scalability analysis may be seen as overkill by developers. This is the case especially if considering that the absence of a scaling assumption represents a possibly infinite range of values to be assumed by the characteristic at hand. Although defining scaling assumptions for all characteristics involves a considerable amount of work, our conversations with computer scientists from both industry and academia suggest that scalability problems often occur because the ranges of values of some domain quantities have been overlooked. The goal-obstacle analysis of the model allows variables and functions for the scalability analysis to be systematically derived from scalability goals, scaling assumptions and agents. This research assumes that a standard goal model is available from the requirements engineering phase, or that KAOS is chosen as the requirements elaboration technique if scalability is being considered early in the development lifecycle. However, a partial model could also be constructed in other stages of the development lifecycle, bearing in mind that a full model is more likely to reveal the full range of potential scalability problems.

Limitations of the Method for Scalability Analysis

The method described in Chapter 6 brings together the requirements engineering and scalability analysis techniques. This method is limited in the sense that some of its steps are just initial ideas, which have not been fully developed and validated. For example, the method indicates that system design variables should be derived from agents in the goal model. However, this selection is currently performed in an ad-hoc manner. Another risk is that, without precise guidelines, the model may lose accuracy when translating from objective functions to preferences. The use of requirements prioritization techniques to derive utility functions also needs to be further investigated, as well as refinement techniques that convert goal quality variables into independent variables that can be used in a weighted sum. Alternative ways to model utility functions can also be investigated.

Furthermore, the design of scalability tests—that is, the selection of a representative set of execution experiments to characterize the sample space of system behavior—has been left out of the scope of this thesis. This is an important activity, which should be included in the step "generate raw data and perform scalability analysis" of the method. Finally, when modeling techniques, such as queuing networks, are incorporated into the method, there will be a need for a feedback loop assessing the degree of accuracy of the analysis and updating the model accordingly. As it stands, the method is useful to developers willing to characterize and analyze the scalability of existing software systems (or prototypes), with sufficient data and infrastructure to run the system over the full range of scaling application domain characteristics. Finally, in order to make our method useful for developers in industry, we need a better understanding of how it fits into popular development methodologies, as well as a quantification of its effort, costs and benefits. For example, is the method only useful to build critical systems that follow strict development processes? Could it be adapted to more dynamic and iterative methods, such as Agile? These questions, however, have been left unanswered in our research and should be tackled in the future.

Despite these limitations, the scalability definition, framework and techniques provide a common understanding of scalability, allowing scalability claims to be fully justified and described with sufficient information for others to confirm or dispute these claims. The techniques also encourage a proactive approach to scalability, addressing difficulties that commonly lead to scalability problems, such as the scaling caused by overlooked application domain and system design characteristics, unusual system usage, software errors, and expected application domain scenarios.

8.4 Future Work

In this section, we list a number of possible directions for future work that cover both the limitations of the techniques described in this thesis and the topics that had to be left out of the scope of this research.

Design of test cases: Scalability analysis is a multi-dimensional problem, involving a number of scaling characteristics and system qualities. Designing a small, yet comprehensive, set of tests for exploring the range of system behaviors is an important activity. In the future, we plan to look at deriving scalability tests from goal models. Doing so allows one to check for test coverage more naturally. In KAOS, a set of requirements is complete with respect to a set of goals if all the goals can be shown to be satisfied when all the requirements are satisfied, assuming the environment assumptions and domain properties hold. Therefore, in principle, tests derived systematically from these requirements, can be shown to cover all the goals of the system.

Extrapolation of analysis results: This work relies on the assumption that there is a running system and sufficient data and hardware infrastructure to study with the system over the full ranges of application domain characteristics. This assumption, however, will rarely hold in the real world. Extrapolating the results of a scalability analysis is of undeniable importance. Extrapolation must take into consideration the different scaling nature of independent and dependent variables, and the uncertainties regarding their values, and include a feedback loop to validate the predictions and adjust the parameters of the technique accordingly. Possible techniques include the combination of modeling and testing. Inevitably, there will be errors in the extrapolation, but it should at least serve as an indicator to support scalability-related decisions, such as the choice between two alternative designs.

Quantitative agent load analysis: Assessing the likelihood of scalability obstacles requires an analysis of the load imposed by goals on agents. Such analysis must consider when each goal inflicts load on an agent, the different types of capacity in an agent, and how the likelihood of a scalability obstacle varies with the required agent's capacity. Naturally, being a requirements engineering technique, it should not include an assessment of the load imposed by the system design. Such an assessment should, however, be considered later during system design and testing, as well as be considered in the scalability analysis.

Modeling and handling uncertainties and disagreements: Techniques are needed for modeling and reasoning about uncertainties and disagreements concerning scaling assumptions in goal models. Currently, disagreements have to be modeled as distinct assumptions, which is an unsatisfactory solution. One possible technique is to use a parameterized model (and confidence intervals) to highlight the impact of different assumptions on the likelihood of scalability obstacles and on the satisfaction of system goals, as to promote discussions among the stakeholders.

Derivation of system design variables, preferences and utility functions: In the scalability method, independent and dependent variables belonging to the system design are still derived in an ad-hoc manner. Also, more systematic techniques are needed to ensure that the analysis will not lose the accuracy introduced by the goal model in the translation of goals objective functions into preference functions. This thesis also presents initial ideas on using requirements prioritization techniques to derive the utility function to be used in the scalability analysis. Further case studies should be completed to validate these initial ideas.

Selection of scalability strategies: Future work should also develop a technique for selecting a strategy from among a set of alternative scalability strategies. Such a technique should consider the scalability obstacle likelihood and criticality, and the costs associated with each alternative.

Expand the catalog of scalability strategies: This thesis has described a number of strategies commonly used in industry to achieve scalability in software systems. There are probably many more. Future work can expand such a catalog so that requirements analysts have more strategies to choose from when resolving scalability obstacles.

Explore the scaling down of software systems: This thesis defines scalability in terms of variation of system characteristics, not growth. From the perspective of goal-obstacle analysis, the problem of scaling down a system to run on a more restricted environment can be seen in an analogous way to the scaling up problem: the restricted capacity of the agent may not be sufficient to support the load imposed by its goals, configuring a scalability obstacle. Nevertheless, mitigating such obstacles in a scaling down system will require different strategies, such as *eliminate irrelevant functionality*. The problem of scaling down needs to be more thoroughly investigated in the future and specific resolution tactics must be defined.

Beyond load and capacity: Our scalability goal-obstacle analysis relies exclusively on the concepts of load and capacity to assess the satisfiability of quality goals, such as performance, availability and reliability. However, additional quality attributes such as adaptability, extensibility and predictability have a different relationship with scalability. Take for example, the scalability obstacle resolution strategy *adapt agent capacity at runtime according to load*. This strategy may be achieved by means of predicting the available capacity of the system when its load varies, and designing it to reconfigure itself or to be easily extended to support this variation. Adaptability, extensibility and predictability are therefore important attributes of scalable systems. Should these attributes be treated as first-order concepts in scalability or contributing factors? Future work should explore the relationship between these attributes and scalability, as to extend our catalog of scalability obstacle resolution strategies and to create design guidelines to implement them.

8.5 Closing Remark

My interest in the topic of scalability started as a combination of intuition and experience as a professional software developer. As we advanced in the research, I started to really contemplate the depth and breadth of this topic. This thesis is a compilation of four years of research, uncountable discussions with researchers and practitioners, and practical experience. It represents a initial, but relevant contribution to the problem of scalability in software engineering.

The scalability framework allows stakeholders to describe the scalability of a software system clearly, tackling the problem of intuitions, ambiguities and inconsistencies underlying the notion of scalability in computing. It also encourages scalability concerns to be clearly expressed and thoroughly investigated in the development of software systems, possibly bringing into evidence problems that could otherwise be overlooked.

We encourage the computing community to adopt some of the ideas presented in this thesis when building their systems and to incorporate scalability into their analysis of other system qualities. Finally, we encourage the academic community to concentrate more research effort into the interesting and important field of software systems scalability.

Appendix A

Scalability Definitions

Like ourselves, other authors have questioned the meaning and use of the term scalability and have attempted to provide better definitions. This appendix shows a list of definitions of the term taken from the scientific and informal literature. Most attempts, however, represent an intuitive ideal or are restricted to a narrow meaning.

A.1 Scalability Definitions in Scientific Literature

- *"Scalability metrics measure the ability of a parallel architecture to increase the number of processors for solving a problem of a increasing size where the parallelism of a given algorithm has already been effectively exploited."* (Zhang et al., 1994)

- *"Scalability is defined as the ability to maintain cost effectiveness as workload grows"* (Luke, 1993) — Where cost effectiveness is defined as a function of *work* (minimum number of operations required to complete a given computational task), *execution time* and a scaling constant between *workload* and *processing resources*.

- *"The scalability of a parallel architecture is a measure of its capacity to effectively utilize an increasing number of processors."* (Kumar and Gupta, 1994)

- *"[...] scalability: the system's ability to increase speedup as the number of processors increase."* (Grama et al., 1993)

- *"Scalability is a property which exhibits performance linearly proportional to the number of processors employed."* (Sun and Rover, 1994)

- *"Scalability measures the ability of a parallel system to improve performance when the sizes of the program and the machine are scaled."* (Zhang and Xu, 1995)

- *"An architecture is scalable with respect to an IT profile and a range of desired capacities if it has a viable set of instantiations over that range."* (Brataas and Hughes, 2004)

- *"An architecture is scalable [...] if it has a [...] linear (or sub-linear) increase in physical resource usage as capacity increases [...]"* (Brataas and Hughes, 2004)

- *"An algorithm-system combination is scalable if the achieved speed-efficiency of the combination can remain constant with increasing system ensemble size, provided the problem size can be increased with the system size."* (Sun et al., 2005).

- *"Scalability is the ability of a parallel algorithm on a parallel architecture to effectively utilize an increasing number of processors."* (Vetter and McCracken, 2001)

- *"An algorithm is scalable if the level of parallelism increases at least linearly with the problem size. An architecture is scalable if it continues to yield the same performance per processor, albeit used on a larger problem size, as the number of processors increases."* (Quinn, 1994)

- *"Scalability is the ability of a distributed simulation to maintain time and spatial consistency as the number of entities and accompanying interactions increase."* (DoD, 1995)

- *"A scalable simulation is one that exhibits improvements in simulation capability in direct proportion to improvements in system architectural capability."* (Law, 1998)

- *[Telecommunications systems] "must be capable of being deployed efficiently at both large and small scales. Over time it must also be possible to add capacity to either support more users, or enhance the quality of service, or both."* (Jogalekar and Woodside, 1997)

- *"A distributed system is said to be scalable from its old state to a new state, if power per invested dollar can be improved (or maintained constant) by evolving the system configuration."* (Jogalekar and Woodside, 1997)

- *"...information systems should be scalable. Informally this means that the system should easily accommodate higher performance levels, because, for example, the size or geographic dispersion of the set of users changes. At the same time, systems should be scalable in the sense that they can easily be adapted to cooperate with future applications."* (Steen et al., 1998)

- *An architecture is scalable if it is "able to accommodate whatever performance level or number of users necessary by simply adding resources to the system, as opposed to replacing the technology for higher performance. [...] A desirable form of scalability is a resource cost that is at most linear in some measure of performance or usage."* (Messerschmitt, 1995)

- *"Let $\gamma(\boldsymbol{a})$ be a function in attribute values **a** returning values in the same unit as $Cost_A$ Furthermore, let $\delta(\boldsymbol{a})$ be a function in **a** returning values in the same performance measure unit as $Perf_A$ with $\forall$ **a**::$\delta(\boldsymbol{a}) \leq$ and $\delta(\boldsymbol{a}) = 0$. An application A is scalable in attribute $Attr_i$ for values up to a maximum a_{max} with respect to γ and δ if the following properties hold: **P1**: A can accommodate values $\boldsymbol{a}_{ref}[i]$ ¡ $a \leq a_{max}$; **P2**: $\forall$ $\boldsymbol{a}_{ref}[i]$ ¡ $a \leq a_{max}$ $\exists$ **r**::$Perf_A(\boldsymbol{a}_{ref}, \boldsymbol{r}_{ref})$ - $Perf_A(\boldsymbol{a}_{ref}(i{:}a), \boldsymbol{r}) \leq \delta(\boldsymbol{a}_{ref}(i{:}a))$; **P3**: $\forall$ $\boldsymbol{a}_{ref}[i]$ ¡ $a \leq a_{max}$:: $Cost_A(\boldsymbol{a}_{ref}(i{:}a), \boldsymbol{r}_{min}(\boldsymbol{a}_{ref}(i{:}a)))$ $\leq \gamma(\boldsymbol{a}_{ref}(i{:}a))$."* (Steen et al., 1998)

- *"[Scalability] denotes the ability to accommodate a growing future load."* (Bahsoon and Emmerich, 2008)

- *"Load scalability [...] the ability to function gracefully, i.e., without undue delay and without unproductive resource consumption or resource contention at light, moderate, or heavy loads while making good use of available resources [...] Space scalability [...] its memory requirements do not grow to intolerable levels as the number of items it supports increases... if its memory requirements increase at most sub-linearly [...] Space-time scalability [...] if it continues to function gracefully as the number of objects it compasses increases by orders of magnitude [...] Structural scalability [...] its implementation or standards do not impede the growth of the number of objects it encompasses, or at least will do so within a chosen time frame [...] Distance scalability [...] if it works well over long distances as well as short distances. [...] Speed/Distance scalability [...] if it works well over long distances as well as short distances at high and low speed."* (Bondi, 2000)

- *"A system is a scalable system if it can be deployed effectively and economically over a range of different 'sizes', suitably defined."* (Jogalekar and Woodside, 1998)

- *"Scalability means not just the ability to operate, but to operate efficiently and with adequate quality of service, over the given range of configurations."* (Jogalekar and Woodside, 2000)

- *"[Def.1] Scalability is the ability to handle increased workload (without adding resources to a system). [Def.2] Scalability is the ability to handle increased workload by repeatedly applying a cost-effective strategy for extending a system's capacity."* (Weinstock and Goodenough, 2006)

- *"Given a reasonable performance on a sample problem, a problem of increased workload can be solved with reasonable performance given a commensurate increase in computational resources."* (Luke, 1993)

- *"Scalability in the context of software engineering is the property of reducing or increasing the scope of methods, processes, and management according to the problem size. On way of assessing scalability is with the notion of scalable adequacy — the effectiveness of a software engineering notation or process when used on differently sized problems."* (Laitinen et al., 2000)

- *"Scalability is the ability of a system to continue to meet its response time or throughput objectives as the demands for the software functions increases."* (Smith and Williams, 2001)

- *"[Scalability is] the ability of a system to keep its performance when the system ensemble size is scaled up."* (Chen and Sun, 2006)

- *[Scalability is the] " ability to provide a basic service to an increasing number of users."* (Scholz and Rouvoy, 2007)

- *"Software scalability is the easy with which the software of a system can be expanded to serve more users and/or work."* (Munsterman, 2008)

- *"Software scalability is the ability to handle increased workload by changing parts of the code (scalability optimization/vertical scalability)."* (Munsterman, 2008)

- *"Software scalability has the ability to be used multiple times in a cost-effective way (environmental flexibility/horizontal scalability)."* (Munsterman, 2008)

- *"Scalability is a measure of an application systems ability to—without modification—cost-effectively provide increased throughput, reduced response time and/or support more users when hardware resources are added."* (Williams and Smith, 2004)

A.2 Scalability Definitions in Informal Literature [1]

- *"Scaling up is the commonly used term for achieving scalability using better, faster, and more expensive hardware. [...] Scaling out leverages the economics of using commodity PC hardware to distribute the processing load across more than one server."* (Microsoft, 2005)

[1] Many of the definitions was collected by Weinstock and Goodenough (2006)

- *"Def.1: [Scalability] is the ability of a computer application or product (hardware or software) to continue to function well when it (or its context) is changed in size or volume in order to meet a user need. Typically, the rescaling is to a larger size or volume. The rescaling can be of the product itself (for example, a line of computer systems of different sizes in terms of storage, RAM, and so forth) or in the scalable object's movement to a new context (for example, a new operating system). Def.2: It is the ability not only to function well in the rescaled situation, but to actually take full advantage of it. For example, an application program would be scalable if it could be moved from a smaller to a larger operating system and take full advantage of the larger operating system in terms of performance (user response time and so forth) and the larger number of users that could be handled."* (SearchDataCenter, 2006)

- *"Def.1: [Scalability is] how well a hardware or software system can adapt to increased demands. For example, a scalable network system would be one that can start with just a few nodes but can easily expand to thousands of nodes. Def.2: [Scalability] refers to anything whose size can be changed (e.g., a scalable font is one that can be represented in different sizes). Def.2: [Scalability,] when used to describe a computer system, the ability to run more than one processor."* (Webopedia, 2006)

- *"A highly scalable device or application implies that it can handle a large increase in users, workload, or transactions without undue strain. Scalable does not always mean that expansion is free. Extra-cost hardware or software may be required to achieve maximum scalability."* (Farlex, 2008)

- *"[Scalability is the] ability to easily change in size or configuration to suit changing conditions. For example, a company that plans to set up a client/server network may want to have a system that not only works with the number of people who will immediately use the system, but the number who may be using it in one year, five years, or ten years."* (ComputerUser, 2008)

- *"Def.1: A system whose performance improves after adding hardware, proportionally to the capacity added, is said to be a scalable system. An algorithm, design, networking protocol, program, or other system is said to scale if it is suitably efficient and practical when applied to large situations (e.g. a large input data set or large number of participating nodes in the case of a distributed system). Def.2: A scalable online transaction processing system or database management system is one that can be upgraded to process more transactions by adding new processors, devices and storage, and which can be upgraded easily and transparently without shutting it down. Def.3: A routing protocol is considered scalable with respect to network size, if the size of the necessary routing table on each node grows as O(log N), where N is the number of nodes in the network."* (Wikipedia, 2008)

- *"Scalability refers to the component's ability to adapt readily to a greater or less intensity of use, volume, or demand while still meeting business objectives."* (Chiu, 2001).

Appendix B

Industry Cases

In order to gain a better understanding of common causes for scalability problems in industry, we conversed with 16 companies of different sizes, industries and maturity levels. This appendix describes our findings and presents a summary of each industry case.

B.1 Motivation

In addition to review the state-of-the-art in scalability research, we sought to gain a better understanding of the treatment of scalability in industry. We were particularly interested on how scalability was dealt with during the development lifecycle, common causes for scalability problems and lessons learned by professional developers who have experienced scalability problems on their systems. For such, we conversed with 16 companies of different sizes, industries and maturity levels. Conversations, took the form of phone or face-to-faces interviews, each lasting around two hours. Interviews were recorded and transcripts were produced. Most interviewees were experienced software developers, who had been closely related with the system they choose to describe.

The interview was designed to uncover the developer's experience and involvement with the system being described, relevant information about the systems functionality, application domain and architecture, the software development process followed, the scalability problems faced and their solutions, and lessons learned from the experience. Towards the end of the interview, there was a couple of "speculative questions" that investigated the interviewee's opinion regarding scalability problems in general. Questions were revised by a psychologist to minimized the chances of biasing the interviewees answers. They are listed in the end of this section.

The interviews, which were conducted in 2007, covered both recently developed systems as well as systems dating back to the 90's. The fact that we have included older systems may raise a concern regarding the validity of our observations for today's reality. However, among the 16 companies, there were three consultancies specialized on performance and scalability. With these consultancies we discussed scalability in more general terms. We asked about commonly found scalability problems, their causes and resolutions, as well as their advice to build scalable systems. The conversations with these consultancies confirmed our own observations from the interviews. These observations are covered in the next section.

While these results were not gathered by any scientific means, it helped us to confirm our intuitions and develop a better understanding of the treatment of scalability in practice. Conversations also provided us with a number of examples used throughout this thesis to motivate discussions, and illustrate common scalability problems and resolutions tactics. In particular, these interviews confirmed the importance of the requirements phase in preventing scalability problems and helped us to articulate common concerns that illustrates the benefits of applying goal-oriented requirements engineering for eliciting scalability requirements.

Interview Questions

1. Can you please quickly describe your experience as a software developer? E.g. Programmer, architect role? Years of experience?
2. What was your involvement in the system you are about to describe? E.g. Design, implementation, testing?
3. Could you please describe the system and its intended functionality?
4. Could you please quickly describe the process used to build the system? E.g. How the requirements and design were captured? What kind of analysis and testing were performed?
5. Could you please describe the application domain of the system? Hardware, environment, users, workload?
6. Could you please describe the main aspects of the system design?
7. Could you please describe the scalability problem faced by the engineering?
8. When did the company realized the system had scalability problems?
9. Could you please describe how the problem was corrected?
10. Was the problem expected or did it take the engineers by surprise?
11. Looking retrospectively, which design and environment characteristics most contributed to the problems? Which system qualities were mostly affected? Can you tell me the scaling range and hard bounds? Can you currently describe the relationship between them?
12. If the problem was expected, how was the threat been dealt with during the requirements, design or testing? Why do you think the problem still happened, despite of all the measures taken?
13. Given enough time and money, what could the software engineers have done to avoid the problem?
14. Under the circumstances at the time, what could have been realistic done to avoid the problem?
15. In your opinion, why so many systems encounter scalability problems?
16. In your opinion, which phase of the software development lifecycle is the most important in order to build scalable systems? E.g. requirements, design, coding, testing?

B.2 An Overview of the Industry Cases

Interviews covered a variety of systems, from small bespoken applications developed in an ad-hoc manner, to complex systems built according to a strict software development process, to infrastructure systems whose main objective was to provide scalability. All of them faced scalability problems. Sometimes, problems were uncovered during load or stress tests. Other times, they were predicted by monitoring the system's load and resource usage during production. However, often they were discovered by customers during system usage.

Scalability problems took developers by surprise on many occasions. Some problems, however, were expected and deliberately not dealt with until they became a reality. Reasons for doing so varied from the belief that developers would be able to solve the problem later easily, to conscious decisions to focus on more pressing issues. Often, scalability problems were fixed through workarounds—up to the point the system became unmanageable, requiring a complete redesign. Sometimes, they were treated as bug fixes to be addressed in the next release cycle. On other cases, the companies chose to let these problems happen, mitigating the consequences every time.

Causes for scalability problems were varied, some of which are listed below. When enquired about the phase of the development lifecycle that was more important when building scalable systems, most interviewees elected requirement elicitation.

Time-to-market pressures: The most cited cause for scalability problems. Time pressure happened for a variety of reasons, such as an upcoming marketing campaign, the need to overcome competitors, and end user pressure. As a consequence, nearly everything has happened. Processes were abandoned, companies felt obliged to squeeze the phases of the software development lifecycle, sometimes removing them altogether. Load tests were moved into the production, management pushed for a quick solution or more features even when scalability problems had been predicted, and systems were released with known scalability problems. As a result, people tended not to take responsibility for scalability failures.

Start-up companies were the most susceptible to such pressures. They were forced to concentrate on functionality in order to gain market share and generate revenue to survive. These companies were often under-staffed, had less experience, and little money for investing in tools and infrastructure. There was generally a belief that the customer would trust the company to fix scalability problems later. It has been argued on a few interviews that a company must get to a certain size before it is able to create a product properly.

In summary, developers were under too much pressure to consider scalability carefully, and when looking back they often believe they did their best under the circumstances at the time.

Complete lack of requirements: Some systems were built without a requirements specification. They were novel products or research tools that did not existed on the market. For this reason, developers implemented features they considered useful. These features were often implemented in an ad-hoc manner and without any concern for scalability.

Lack of non-functional requirements: On many occasions, a requirements specification existed, but it concentrated on functional requirements, never considering load, growth characteristics or technical and physical boundaries. Scalability requirements were often overlooked or undervalued, following the dangerous belief that scalability could be addressed later. Without such requirements, some developers declared they deliberately did not address scalability with the fear of creating new problems or of optimizing something that was not required. As a consequence, scalability was not considered when designing some systems. There were also cases when non-functional requirements were only elicited after scalability problems became apparent.

Some companies raised scalability questions from the outset. These were companies whose business was to provide scalability, whose revenue was driven by volume processed (such as telecom companies), who by nature dealt with extreme high volume (such as scientific computing) and companies in more mature industries (such as banks)

Imprecise non-functional requirements: Sometimes, scalability requirements were drawn naturally from QoS agreements with respect to different values of the scaling ranges. This is true particularly for some sectors, such as telecom. Other times, however, scalability requirements seem to have been derived from folklore or the willingness to comply with generally accepted good practices in software development, which could impose unjustifiable demands on the system design. This is the case of the vague belief in the need for linear scalability — e.g. when throughput is required to increase linearly with machine capacity (Brataas and Hughes, 2004).

There were also occasions where the application domain was not known. For example, when developing novel products whose market did not exist and whose popularity was hard to predict. On these cases, imprecise requirements were sometimes elicited from informal sales conversations or were based on wrong assumptions. It was also observed that people may know the market, but were often bad at estimating concurrency and growth. In some cases, eliciting precise non-functional requirements is difficult because they are very dependent on the complexity and distribution of the customer data, not only the load.

Scalability requirements were often described in terms of boundaries. Focusing on limiting cases of relevant domain characteristics may lead to a solution that does not perform at its best for the sub-ranges of greatest interest to the stakeholder. For example, a system that is optimized to achieve the best performance at peak load may not be optimal under normal load, which may be more important to stakeholders. Another observed cause for imprecise requirements was little interaction of the commercial team with the technical staff, leading to requirements that were not technically feasible.

Changing non-functional requirements: Requirements also changed during the lifetime of some of the systems discussed. Often because they were found to be incorrect or because the application domain had changed. That were also cases in which the company targeted new markets, whose scale had a very different nature from the markets the system was originally built to address.

Another common case of changing non-functional requirements was on licitation processes, where the real numbers were only disclosed after the contract was signed.

Inadequate design: Give a lack of non-functional requirements, some systems were built without scalability in mind and architects chosen inadequate solutions to the problems (e.g. the use of synchronous communication when there were many concurrent requests). On more extreme cases, there was a complete lack of architecture.

Scalability through hardware seemed a popular strategy. Some companies hit the limits of vertical scaling. Although horizontal scaling was often used, it was not always properly implemented. A few companies, for example, scaled the system as a monolithic instance, not taking full advantage of the hardware infrastructure. It is however recognized that some problems are more serializable then others, making some systems easier to design for horizontal scaling. This is the case with telecom and some scientific applications.

Inexperienced developers: Interviewees often described developers as inexperienced—with large systems, with a particular industry, with the environment being used, or with software development in general. Some companies relied on trainees with little experience and keen to use new technology.

Basic implementation mistakes: Perhaps the little experience can justify some of the mistakes made when building the software systems described on the interviews. Developers were reported of ignoring basic computer science concepts, having little interest about the computational complexity of algorithms, designing interfaces without concern for performance, using wrong design patterns, and so fourth. Some of the problems described were well known by the software engineering community, indicating that a proper research into the adopted solution was not performed.

Inadequate tools and infrastructure: Another cause of scalability problems often mentioned was the use of inadequate tools and infrastructure. The reasons varied from lack of money to internal organization rules (e.g. bureaucracy to acquire proprietary tools or new machines).

Inadequate use of technology: The use of new technology, without sufficient experience of the developing team, was blamed for the creation of code with performance and scalability problems. That included bad configuration of tools, from-scratch-implementation of services already provided by the technologies, and the improper use of technology (e.g. not taking advantage of sets when using a relational database).

Lack of load/stress tests: Load and stress tests were described as overlooked and undervalued, both by developers and customers. They are activities that were often ignored due to time-to-market pressures or wrong beliefs about the system capacity and ease of change. On a number of cases, load and stress tests were only performed after the system had been put into production and had presented scalability problems. Mentioned exceptions were systems developed for restricted environment, businesses whose revenue are driven by volume, and systems whose business is to provide scalability. It has also been argued by some interviewees, working as performance and scalability consultants, that few industries understand the importance of non-functional tests, that testing is seen as an under-job, and that there are few testing specialists on the market. Load and performance tests were described as very time consuming to plan and execute.

Inadequate load/stress tests: There were cases in which load/stress tests were performed before production, but not in an adequate manner. Sometimes, tests were performed with much less load then the system would face in production or without considering future growth (e.g. only with the average load or with the load generated by the company's biggest customer *at the time).* Other systems were tested in a different infrastructure/topology from the production environment (e.g. locally and without taking the network latency into consideration, or in a simpler network).

Inability to perform load/stress tests: Some interviewees argued it was impossible to perform load/stress tests. Reasons cited were insufficiently powerful infrastructure, and inability to simulate production load (e.g. Internet-based applications).

Inability to reliably predict scalability beyond tested load: All consultants mentioned the difficulty in extrapolating test results to heavier loads or different environments. Difficulties were explained by the large number of variables and assumptions. In fact, two of the consultancies routinely ask for equivalent testing infrastructure from customers or do not guarantee results, explaining they are based on assumptions that could be incorrect.

Once-only load/stress tests: The importance of continuous functional testing is well understood and accepted in the software industry. However, load tests were rarely repeated after changes to the systems described. As a result, scalability problems were sometimes caused by knock-on effects of these changes.

Dangerous beliefs: We have witness a number of dangerous beliefs that led to some of the causes listed above. For example, some developers admitted to have assumed some piece of code would simply work or have linear scalability without having tested for it. There was also a general reactive approach to scalability, in which developers believed that systems evolved 'naturally' and scalability risks could be easily addressed when they became a reality. Normally, developers trusted a more powerful machine or more machines would solve scalability problems, and some believed that scalability through hardware had no limits. It was also believed that the customer would generally trust the company to fix scalability problems and that it was more economically viable to get consultants to solve scalability problems rather than building scalability into the application. It was also pointed out by consultants that developers believe what is advertised in terms of volume of requests handled by application servers and alike, forgetting that the system performance and scalability also depends on the design of their applications.

Unplanned evolution: Another major cause of scalability problems was unplanned evolution. Many systems evolved from bespoken applications or research tools to commercial products. They were, for this reason, not designed to handle the scale they ended up facing. Engineers struggled to accommodate the variability of requirements from different customers, making use of workarounds that would often lead to scalability problems. Interestingly, we have observed two opposite scenarios with research tools, those whose scalability had been completely neglected and those whose scalability had been thoroughly investigated—it basically dependent on whether scalability was part of the problem the research was trying to address. Finally, there were also the systems who changed markets due to a business opportunity, suddenly having to deal with a complete different load.

Unexpected scaling: Developers were also often taken by surprise for the scaling of some aspect the system that had not been considered. For example, developers might have planned for the scaling of some characteristics, but overlooked others (e.g. consider the number of incoming request, but not the size the data these requests were carrying). Unexpected scaling was also caused by unpredicted usage scenarios, software errors and knock-on effects of software/infrastructure changes.

B.3 Summaries of Industry Cases

This section contains a summary of the interviews conducted as part of this research. When the interviewee told us the story of more than one system, one was selected to be described here. Details that could compromise the identity of the companies were concealed and the interviewees names are fictitious. Some cases could be described with more details then others, as authorized by the interviewees.

The description of industry cases cover the following points: (1) the interviewed experience and role on the system being described, (2) a brief description of the system and of its architecture, when relevant, (3) the development process followed, (4) the scaling characteristics of the application domain, (5) the scalability problems faced, when they became apparent and how they have been solved, (6) lessons learned, and finally, (7) their opinion on issues related to scalability in general.

B.3.1 ERP System

Inacio has over ten years of experience as a professional programmer and software architect. He has been involved in the design and implementation of a number of systems. His role in the system described was of technical leader. He was also responsible for the optimization of the application's database.

Inacio described a client-server ERP system built in the 90's. The system was divided in OLTP and OLAP. The OLTP module was always implemented first for customers. The system used a SQL client that connected directly to an *Oracle* database. The company started by developing bespoken systems, but later evolved to a single product that fulfilled the requirements of all customers.

The company did not follow any particular development process. There were two main distinct groups: the account managers, who were responsible for the business contracts and requirements elicitation, and the engineers, who were responsible for implementation, testing and technical support. Initially, only functional requirements were elicited. The company followed no particular testing strategy. Once the system passed very basic functional tests, it was deployed at the customer site. Particularly, no workload testing or system optimization was done before the system went into production. The company counted with two implicit assumptions: (1) The database would be able to handle the load when OLAP was added, and (2) in the worst case scenario, an upgrade of the infrastructure would solve any scalability problem.

The company's customer base, composed by banks and medium size companies in Brazil, grew from two to nearly one hundred in three years. In average, the deployed system would have ten to fifty concurrent users, possibly reaching one hundred. Query frequency depended on the type of the user: humans or external systems. In average, 80% of human users used OLTP, performing small queries at a rate of one to five queries per minute; the remaining 20% used OLAP, requesting ten reports per day. Although the majority the reports had one or two pages, some reports could reach three thousand pages. External systems sent, in average, ten thousand invoices per day, each with ten lines. Most operations were performed during working hours.

The company faced a number of scalability problems. As the company's customers-base increased and the system evolved from bespoken applications to a product, the engineers struggled to accommodate the variability of new requirements. The system could not easily support the addition of new types of queries, suffered to integrate with external systems due to the import/export of large data sets, and could not handle the increasing number of concurrent users, query load and reports sizes. Attempts to scale exhausted the system resources and took unacceptable times.

Due to the lack of workload testing, problems were normally uncovered by customers. As the first complains were raised, engineers realized the system had scalability problems, but did not understand its magnitude. Engineers then requested account management to characterize the application domain. However, conflict of interest between engineering and management meant that this task was poorly executed—without detail or precision, and only in terms of the load at the time. Management felt it was the customers responsibility to provide them with the characterization of the application domain, and was more concerned in quickly delivering the system to generate revenue. Engineers felt it was the management responsibility to provide them with precise requirements and the time and resources to tackle the problem. No one accepted the responsibility for the failure. The scalability problems were never properly corrected, instead the company implemented a number of workarounds, particularly in terms of database optimization.

Inacio blames the scalability problems at the lack of a process that treated scalability and emphasized that such a process should not only provide the ability to quantify and qualify the scalability characteristics of the system, but should also assign responsibilities to all involved. He believes that an *Agile* development method would have been more appropriate and that non-functional requirements should have been elicited earlier.

When enquired about scalability problems in general, the interviewed emphasized the importance of carefully described requirements. He believes that, if not given proper requirements, architects and programmers will either overlook the non-functional aspects of the system or will deliberately not address them with the fear of creating new problems for trying to optimize aspects that were not required.

B.3.2 Message Validator

Mark has been a software developer for twenty years, seven of which he has worked as an independent adviser and technical leader. The system he describes was first developed as part of his PhD research and turned into a commercial product.

The system validates messages against a number of validation rules written by the customer in a graphical tool. Its engine is commercialized as an API, which is used by customers to build their own systems—normally standalone applications or Web services. When the system starts, rules are loaded in the engine, which then waits for messages to validate. During validation, the message is loaded entirely into memory, so it can be randomly accessed. Rules run independently and the engine may hold a number of messages in memory.

The system was first written as a prototype to solve a research problem. It then improved in order to demonstrate its usefulness in a real-world setting. By the time it became a commercial product, the system had gone through a lot of performance optimization and had received an user interface. There has never been a requirement specification. Mark initially implemented features he considered the customer would want. Load estimates came often from informal sales conversations and, in this case, were highly exaggerated. As a result, the system could handle a much greater number of messages then it has ever been required. The system has 70% of unit test coverage. Performance figures with realistic examples were produced to support the sales process.

The system has been used in a number of different domains. Currently, most of its customers are in the finance sector. The number of rules developed by customers varies greatly. The largest set of rules the system is known to be used with is 2,5 thousand. Size and complexity of messages being validated also vary, normally between 200 KB and 100 MB.

Recently, Mark has been approached by a company interested in validating 2 GB documents. The relationship between validation time and document size had been investigated as part of the PhD research. The system's inability to validate large messages is a known scalability issue. Mark had tried to address this problem during the PhD research, but he did not succeeded. When the system became a commercial product, they chose not to address this problem and concentrate instead on functionality and performance. This limitation has never been a problem to the company, as the system was normally sold for transaction processing. At the time of the interview, Mark had not yet decided whether they should change the system to support larger documents. The prospective customer was aware of the limitation and conversations were taking place.

Mark does not regret his decision of prioritizing functionality and performance. He explains that if a new product does not have enough features, it fails immediately. However, if it has scalability problems, customers will normally trust the company to fix it. Mark's advice is to understand the system's scalability from a theoretical point of view, but optimize it to the typical case, rather then try to fix it to extreme scenarios. He believes it becomes important to understand the system behavior beyond the common cases if there is a chance that the system will need a complete redesign when limits are reached.

B.3.3 Cluster Infrastructure

Steve has over twenty years experience in the software industry. He has described the development of a cluster infrastructure, for which he has been one of the principle software architects. The infrastructure was first built by a start-up company on the 90's. Given that a cluster is aimed at providing scalability, this quality was a major concern. In particular, architects were concerned with the amount of coordination and synchronization they would implement between servers in the cluster and the provision of ACID guaranties. They deliberately decided to provide a weaker coordination between cluster members and relaxed ACID guaranties in order to gain scalability. Retrospectively, these proved to be good decisions—the cluster achieved linear scalability at least in the tens of servers.

As is common with start-up companies, particularly the ones building a system to which a market does not exist yet, there was no requirements gathering. The development team concentrated on implementing what they believed to be a suitable set of features and on achieving good scalability. The company was also under-staffed, both code and tests were written by the same developers. As a consequence, there were never enough tests and functional tests were given priority over scalability ones.

The cluster was launched without any known scalability limits. It was not until one year later that a scalability problem became apparent, taking the architects by surprise. The cluster provided an advertisement feature for services. At start-up, every member of the cluster would advertise each service individually. As customers built bigger and more complex applications, the number of services in the cluster increased, generating a large number of announcements that led to unacceptable booting times. The problem was promptly corrected.

During development, the architects were constantly concerned with the number of servers and the load the cluster could support, rather than the number of services that could be deployed on each server. They had, therefore, tested the cluster with much fewer services than were later created by customers in the field. Steve believes this scalability problem could have been predicted if this particular scaling characteristic had not been overlooked.

The company was later acquired by a larger organization, adopting a more rigorous software development process. The process included a dedicated quality assurance team that would write functional and load tests as the code was being developed. Yet, there were no release requirements in terms of scalability. If a customer complained that the cluster was not scalable in some dimension, the problem would be fixed in the next release.

Steve believes that achieving software scalability requires a proactive approach, rather then a reactive one. Developers should ask customers about the scaling aspects of their applications and try to project the limits the system may hit. He also emphasizes the importance of repeating load tests prior to releases and negotiating software quality trade-offs among project managers, software developers and customers. Nevertheless, the interviewee alerts that there are many other difficulties, such as testing Internet-scale applications under realistic load and the pressure for developing features suffered by start-up companies to gain market share. Finally, Steve believes that a methodology to treat software scalability would be beneficial, yet he points out that there is a lot of discussion on the software engineering community about how and when companies should introduce more rigorous software development processes.

B.3.4 Telephone Billing System

Daniel has over ten years professional experience in the software industry. He has worked with a number of programming languages, databases and operating systems. He is currently a project manager. Daniel describes the development of a large telephone billing system. When Daniel joined the project, 80% of the system functionality was implemented. However, he estimates that only 15% of the code that currently provides the system's availability and scalability had been written.

The system is responsible for the billing of telephone calls. An telephone call is composed by a number of messages that contain a series of information, such as the caller and callee identities, the duration of the call, and who terminated the call. The system captures these messages from the telecom operator's network, identifying each message, and reconstructing the call before it is saved on the database to be made available to a number of business applications. The system is composed by modules responsible for distinct tasks such as identifying, ordering and decoding messages, and composing calls. The system has to deal with tens of thousands of messages per second, with over a hundred different message types that are mapped to a number of distinct protocols. The resulting call is composed by over 100 fields.

The company was chosen by the telecom operator through a licitation process. For this reason, both functional and non-functional requirements had been clearly defined. The request for proposal (RFP) document included a number of international and governmental norms the system should comply with, as well as the specific requirements from the telecom operator. At the time, the telecom operator planned to expand its business and provided the current and the maximum expected load, as well as the time frame between them. The project team was composed by experienced software developers in the sector. They were particularly concerned with memory and disk usage, which had to be estimated through prototyping tests for the commercial proposal.

After the licitation process was won, more details about the application domain were disclosed. Even though the project team had inflated the original numbers provided by the telecom operator, the actual load was greater than expected. In the system, large part of the processing was done in memory and all messages, logs and events were stored in disk. Additionally, in order to prevent messages from being lost on the network, the system kept each message on a memory buffer until the processing was complete. With the new load estimate, both memory and disk would face exhaustion. Correcting the problem required a major refactoring, with happened before the system was put into production.

The system was built with availability and scalability in mind. Tests during the development process were limited due to the difficulty of simulating a real production load in the telecom sector, including the volume of messages, the variety of protocols and their relationships. Both functional and non-functional tests were repeated after the system refactoring. When completed, the system was tested in the actual network with real load. Only then, it was put into production. The company constantly monitors the system. Although the system has never halted due to scalability problems, it has on rare occasions produced inconsistent results due to an extended network downtime and, consequently, an overflow of the memory buffer. They have now migrated the system to a 64 bit machine, increasing the address space.

Daniel admits that the team faced many problems during the system development, mostly due to the lack of a formal project management, which they are now trying to address. He attributes the success of the project to the team's experience in the telecom sector and a proactive approach to uncover and resolve scalability problems.

When enquired about scalability problems in general, Daniel blames the lack of precise requirements and the little importance given to non-functional tests.

B.3.5 Mobile Services

Andrew has been a professional developer for seven years. He tells the story of a start-up company that specialized in mobile applications. Andrew has been involved with the architecture, implementation and testing of the applications described. More specifically, the company provides entertainment and information services on mobile phones through technologies, such as Short Message Service (SMS), Multimedia Message Service (MMS) and Wireless Application Protocol (WAP). All applications are similar in nature, receiving messages from telecom operators, processing these messages and sending a response. The company has lately developed an integration platform that allows third party systems to connect to the telecom operators.

Historically, the company developed bespoken systems, which were later adapted and sold to other customers. In the early days, applications were developed in an ad-hoc manner, some as a monolithic block. These applications regularly suffered from scalability problems, which were usually corrected through workarounds. Problems were often caused by insufficient knowledge of the technologies being used and a complete lack of process. The technical team had predicted many of the problems, but they were under constant pressure from the business area to create new applications and maintain existing ones. Some products, with nearly identical functionality, were implemented by distinct teams with different architectures and technologies. The company was incurring high costs to maintain these products, when a single software would have been sufficient. The company was also loosing talents and lacking staff to create new products.

As the company matured, they have drastically changed their vision, aiming to develop software that can be reused to distinct customers. In the eight months prior to the interview, they started to follow a more rigorous process. Business, technical and infrastructure teams now work together to model the products as set of independent and reusable services that can be replicated to achieve the required scalability. The code is instrumented and the services are constantly monitored. One difficulty is that the company is launching new products into the market, for which the public acceptance, and therefore the load, are hard to predict. For this reason, they currently focus on building application that can take advantage of horizontal scaling.

Nevertheless, Andrew still considers the testing process far from ideal. Functional tests are performed during development. Basic non-functional tests are only performed during deployment. Once the system has gone into production, changes are normally made without repeating the tests. An exception is the integration platform they have recently built, which has been more thoroughly tested because of QoS agreements.

When enquired about the major causes of scalability problems, Andrew blames the insufficient knowledge and inexperience among developers about the technologies being used. He also believes developers are overlooking basic computer science concepts. Companies need better professionals and an alignment of interest between business and technical groups, Andrew concludes.

B.3.6 Web Application

Colin has been a professional developer for seven years. He is a certified database administrator (DBA) and specializes in managing Web and e-mail servers. He also does consultancy work. He has joined his current company in 2004, where he is responsible for managing the hardware infrastructure and Web servers.

The company envisions and develops new Web-based products to be launched on the market by their customers, telecom operators. As they specialize on novelty applications, their popularity are hard to predict. Expected load is rarely discussed. Instead, they have a scalability strategy to support a varying load. Their SLAs are only in terms of availability, never performance.

Applications have similar architectures—a number of Web servers running on commodities machines to which the load is dynamically distributed. They scale horizontally, by adding more hardware as the demand for a particular application increases. This strategy is built into their business model. New applications are normally deployed in shared servers, and may receive a dedicated server if they prove to be a success. Load analysis is performed weekly, sometimes daily, and the number of servers is adjusted according to the demand. They also maintain a few idle machines to support an increase in the load due to a marketing campaign or some other reason. In the same way a product may have a sudden popularity, it may quickly decrease, as when an offer gets to an end. For this reason, the company cannot do a large upfront investment in hardware.

Historically, the company has never built systems to be scalable. It was assumed that the need arising, they would find a solution to scale. Developers have done stress tests on applications, but forgot to account for all the scaling characteristics. For example, on the past, they have considered the number of requests, but overlooked the size of the data being received. As the system started to support high resolution images and videos, the machine limits were reached with fewer requests than had been predicted by the stress test.

Colin described a recent availability problem. A change in the cache implementation slowed down the operations involving the database. This delay led users to retry their operations, causing a sudden increase in the number of requests and bringing the system down. He admits developers had not expected such a problem, but admits that if the company had to prepare for such situations, they would have to maintain a very large number of idle machines. The interviewee considers as their main scalability problem the cost of monitoring and managing all these machines, an activity that is very time consuming. He believes the true limitation is financial. Could they afford commercial Web servers and more powerful hardware they would not have scalability problems, because the supported load is informed upfront by the vendor.

When enquired about the major causes for scalability problems, Colin estimates that in 90% of the cases, the problem is on the code, not the architecture. Projects are usually under tight deadlines and insufficient money. Designing for scalability requires a high investment in time and money. He speculates that it is more economically viable to hire a consultant to solve scalability problems than try to build scalability into the application. Thinking along the same lines, the interviewee adds that it is not worth to spend too much time eliciting scalability requirements. Instead, developers should concentrate on building an application that scales through hardware and carefully choose their supporting software, such as Web servers and databases. Colin believes hardware scaling has no limits, particularly in the Web world and considering today's technologies, such as servers virtualization. He also admits he would not specifically test for scalability because, at least in the Web world, it is reasonable to assume it is linear. This may not be the best solution, but it is cheaper, he concludes.

B.3.7 Funding Compliance System

Iracema has ten years experience in the software industry. She has worked with a number of programming languages and technologies. She was the a technical leader of the project being described and responsible for the system architecture.

The interviewee works for a governmental institution that funds national businesses. In exchange for funding, the institution requires the compliance with a number of rules. Occasionally, inspectors visit randomly selected business to evaluate compliance. All aspects of the visit are logged into a system, such as any irregularity found, documents requested and the decision to continue or cancel funding. The system then generates a partially filled Microsoft Word report that is edited by the inspector and checked back for eventual reference.

The system, which is now in its third incarnation, was developed in-house. It was the organization's first Java project. They chose to use an open source application server and a persistence framework to communicate with five different databases.

The circumstances were far from ideal. The system was being built to support a process that had not been fully defined. They were, therefore, aiming for a moving target. For internal reasons, the project team had no control over the choice of infrastructure in which the system would run. In fact, development started without the infrastructure had been decided on. The team first tried to follow the RUP process, but even before getting to the implementation phase, pressures from the end user made developers abandon the process. The first version of the system was then built in an ad-hoc manner, based only on functional requirements. When the system went into production, the end user was not satisfied with implemented features or the system's performance. The second version started almost from zero. This time, developers imposed a minimum process, which included use cases, sequence diagrams and test cases. The system entered the User Acceptance Tests (UAT) phase, which concentrated mainly on functionality. Performance was only measured subjectively, as perceived by the user playing the role of the tester. The third version expanded the system's set of features.

Some problems became apparent when the number of concurrent users increased during production. For example, individual queries that during UAT were completed almost instantaneously, started to take unacceptable times when executed concurrently. This particular problem was due to insufficient knowledge of the Java persistence framework being used and was corrected through reconfiguration. The system has also run out of disk when the users started to create reports which were much larger than expected. These large reports were considered unnecessary by the organization and the problem was solved by limiting the size of the report that the end user could create.

Looking back, Iracema recognizes a number of mistakes: They had only considered functional requirements. Although they had been told how many users the system would have, they never simulated the production load. In fact, load and stress tests had been completely overlooked. The system had been tested only locally, so they never accounted for the network latency. The team also did not considered the size of the reports or enquired whether other applications would be sharing the servers with their system.

Regarding scalability problems in general, Iracema believes that planning is undervalued. People only invest time and money when a problem occurs. Developers have to work under tight deadlines and with limited money. Another problem is inexperience with new technologies and development of large systems.

B.3.8 ETL System

Mathew has 8 years experience in the software industry. He has described a system that he has helped to migrate from C++ to Java. The system involved a data Extraction, Transformation and Load (ETL) process. Data was extracted from the customer datasources into an object model that was held in memory. The system would then perform a n-1 comparison of all entities to remove duplicates before the data was loaded into the database. These in-memory comparisons were the source of one of the system's scalability problems.

The company was a spin-off from a university and the system evolved from a research prototype into a commercial product. For this reason, its first C++ implementation had not been driven by a requirements specification. A few years later, the company decided to migrate the system to Java. The migration consisted mainly in re-writing the code, while maintaining the original architecture. The new system was tested thoroughly, considering the load generated by the leading players of the finance sector, which at the time did not exceed 4GB of input data. Tests were always performed on a hardware infrastructure that matched the customers production environment.

The company that historically targeted the financial sector, attempted to apply their solution to a different problem space. A contract was signed with a data aggregator who aimed to combine different datasources into a homogeneous set of data. It was initially agreed that the system would have to load 60 GB of data. The technical team suspected the ETL process would take a while for that amount of data, but assumed it would work. As the data extractors started to be built, developers realized that the process was taking much longer than expected. Meanwhile, more datasources had been incorporated to the contract, and the system would have to handle 300 GB of data. An extrapolation performed by the developers indicated that doing so would take no less than 12 years. The ETL process was then re-designed, leaving all the deduplication and disambiguation to the database. The new implementation improved the system performance in over 200%, achieving an acceptable processing time. As with the other customers, tests were performed in a hardware infrastructure that matched the production environment.

Looking back, the interviewee concludes that the story could not have been much different. He observes that this particular ETL problem could have been avoided if a DBA had been involved in the early days of the company—databases are optimized to do bulk comparisons. Nevertheless, as with many start-up companies, the system evolved from bespoken projects to a product. Although, other forms of data extraction had often been discussed, the priority was always closing new business deals and developing new functionality. The interviewee also explained that scalability requirements were particularly hard to define. Processing time depended not only on the amount of data and number of concurrent users, but also on the complexity of that data. Customers, themselves, rarely expressed performance concerns.

Matthew considers that software development for scalable systems has become much easier over the last few years. Organizations have now, at their disposal, a number of open source tools to address common problems. He also emphasizes the importance of enforcing good design and programming practices. When enquired about the phase of the development lifecycle that is more important to scalability, Matthew elected requirements, followed by design, testing and implementation.

B.3.9 Service Control System

John has five years experience in the software industry. He has worked on several open source projects and is the technical leader of the system being described. His responsibilities include eliciting requirements, defining the system's architecture, choosing tools and managing developers. When he joined the company, the system was almost fully implemented and was already deployed on a customer site.

The system, which controls over the counter services, was first developed at a customer request. It monitors the counters, distributes tickets, controls queues of people waiting to be served, suggests how many counters need to be active at certain times of the day, and collects feedback from the people being served.

The initial technical team was composed by one professional developer and two trainees. The system was built in a very ad-hoc manner. The project manager elicited functional requirements with the customer, without ever interacting with the technical team. There was not an overall architecture, each developer designed the part of the system he was responsible for. There was very little communication between team members. Functional tests were basic. The system's performance was effectively tested during production, when the first non-functional requirements started to appear.

At the time, the system was deployed in a bank with three agencies and a total of thirty users. It controlled around seven hundred services and two thousand transactions. Some time later, the customer deployed the system in more agencies, doubling the load. As the number of transactions increased, database queries started to take too long. This problem was due to a bad configured object-relational persistence framework. As the system was deployed in geographically distant agencies, the users experienced response delays and deadlocks. Errors were basic, such as badly designed reports and data being transferred without compactation.

The company then started to negotiate with a second bank, with forty agencies and fifteen users per agency. At this point, they started to see the potential of the system to become a commercial product. The company then adopted a development process based on CMMI. They prioritized requirements, reduced the release cycle, started to use a tool to manage requirements and bugs, tried to follow coding best practices and wrote test cases. This time, the customer imposed a few non-functional requirements. With the agreement of the customer, they estimated a future load that was 20% higher than the current one. Yet, load tests were performed locally and only after the system had been deployed. Later, the network latency caused problems.

At the time of the interview, negotiations with an even larger customer were taking place. The company has never considered the possibility to scale beyond their current customers and expect scalability problems. In particular, they will need vertical scaling, which is not supported by the application. They also expect problems with the synchronization of service counters.

Scalability problems usually took developers by surprise. Most problems were corrected with workarounds, but twice they had to change the architecture drastically. There has never been a change that was not triggered by a customer complaint. Looking back, John recognizes that some problems were well known in the software community and that if load tests had been performed, most problems would have been avoided. Unfortunately, developers were overwhelmed with tasks that had been badly planned, he concludes.

John believes that the biggest causes of scalability problems are wrong tools, the application of wrong patterns, little experience of developers with large systems, little concerned with the computational complexity of certain parts of the code, wrong assumptions about the scaling behavior of the system, lack of load tests or system profiling, and lack of non-functional requirements. John elects requirements as the most relevant phase for the development of scalable systems and warns that it is difficult to predict load upfront. Implementation is elected as the second in order of importance.

B.3.10 Field Services Systems

Alan has worked for 11 years in the software industry. As a consultant, he has played a number of roles, including requirements engineer, programmer, software architect, testing engineer and operational support. He is currently working for a company that specializes in field services systems. In the system described, Alan has been involved with the design, development, testing and support.

A field services system helps to manage complex field service operations. It includes scheduling jobs, dispatching workers to specific locations and providing up-to-date and accurate information to workers on the field. The systems developed by Alan's company have, in average, over one hundred users (workers), receiving each five to ten jobs per day through a mobile device. Each device downloads roughly one thousand pages from the server at start up, and another two hundred pages during the day. In addition to that, the system constantly receives GPS information from the mobile devices and performs automatic synchronization. All these messages consume bandwidth and may create queues waiting for processing in the system's back-end server.

Alan explains that the system is heavily customized for different fields. The development process usually involves a Request For Proposals (RFP) by a company wishing to buy an field service solution. The RFP contains the functional specification and a few non-functional requirements, such as the number of users and mobile devices. When competition is won and the project starts, there is usually another round of requirements elicitation. At this stage, the company agrees with hardware and network requirements, response time, load to be supported by each release, and how far the system should scale. The company then creates the technical architecture, functional specification and a system integration plan. The system is developed by concurrent workstreams: implementation, hardware and network, and integration. Testing includes unit test, system test, integration test, stress test and UAT tests. The customer is normally involved in the tests.

Some scalability problems are discovered during the stress test and others avoided by monitoring the system's network. However, most scalability problems appear after the system has been put into production—usually because the amount of data received by the system exceeds the company's expectations. More data means more bandwidth usage, longer message processing times, increased queues size and longer end-users waiting times. Devices are then forced to reconnect, spending more bandwidth and incurring higher costs to the customer.

Alan explains that these problems happen because these systems cannot be tested at the production levels. The company normally test the systems with at most ten users and three to four jobs every few hours. Simulation cannot take into account the GPRS network, he clarifies. In addition, systems are heavily customized, use different mobile devices, process different data and are deployed on different networks. Differences are too great to directly transfer the experience with one customer to another. For this reason, scalability problems are always expected. Alan does not see how it could be different.

Regarding scalability problems in general, the interviewee believes that projects usually are given less money then necessary. For this reason, development time is reduced, the infrastructure is under-specified, good tools are not used, tests are overlooked and developers are not prepared. Alan believes that the most important phase for producing scalable systems is testing.

B.3.11 Communication Services

Christian has been in the software industry for 13 years. He was one of the original architects of the system being described. In ten years with the company, he has also worked in the system's implementation, deployment, operations support and sales.

The company develops software for telephone operators, telecom operators and cable companies, managing services and equipment for them. The set of services offered by the company is broad, varying from dial-up ISP authentication, to e-mail provisioning, to digital television. The system currently supports over ten million users and over fifty million transactions a day.

The system, which is now in its fourth incarnation, was originally built by five people. Every major change in architecture was driven by scalability concerns. From the outset, there was a scalability question raised. Their first customer had over one million users and was planning to grow ten thousand users a week. It was, therefore, a requirement that the system could grow from one to three million users in three years.

The system's first incarnation did not actually solve the scalability problem. After it went into production, the system started to evolve to contain more services, reaching five different products and one million devices connected to the network. Adding services required a considerable effort in configuration and customization, forcing the system to go offline for long periods. The architecture was static and the data structure was very monolithic. The company then decided to move from a database approach to a object one, building an information model and decoupling business entities into separate objects with a link between them. They could now store objects into different locations, scaling their data repository and co-locating objects to gain performance.

The system's second and third incarnations were built around CORBA. They were using horizontal scaling—when a machine was reaching its limits, another instance of the system would be added to spread the load. That solution was wasteful because the system had two types of end user with very different usage profiles and one requiring a faster service than the other. Replicating a monolithic system would not solve the problem for users requiring a faster service. The company, then, decided to break the system down into services that could scale individually, spreading the load according to the type of incoming request.

The fourth and current incarnation of the system was created because a very large prospective customer had operational requirements—in terms of transaction rates—that the company could not meet. Observing the technology trends in industry, the company opted for building a brand new system based on Java 2 Enterprise Edition (J2EE). This system has been sufficient for their biggest customers, and currently they have no scalability problems.

Christian recalls that over the years, the system faced a number of scalability problems, most of which were expected. They tried to adopt RUP, but felt the process was too heavy for them at the time. He explains that they deliberately chose the quick solution when under market pressure. On one occasion, they had to rebuild their system completely due to performance and scalability problems. Learned that lesson, the company always aimed to build systems with potential to scale. Machines would initially be over-specified and new ones would be added when necessary. Christian's experience is that customers in their industry generally know how much they expect to grow in the next few years. Looking back, Christian concludes, we have done the best we could with the tools we had.

When enquired about scalability problems in general, Christian said that people are normally not aware that they will have scalability problems. They do not understand the problem space and do not realize that requirements are not fixed. Even when people are aware of scalability issues, they believe they will "cross that bridge when it comes to it". The potential for scalability should be there from day one. However, the interviewee adverts, the company must get to a certain size before they can create a product properly. When companies are small, they have to cut corners to generate the revenue they need to survive.

For Christian, requirements elicitation is the most important phase in the development of scalable systems. On his experience, that is the phase that companies on his field have more time for, because it is part of the sales process. Design is the phase that is most affected by time pressures. Performance and scalability testing are often pushed into the field and problems are resolved as bug fixes in the next release cycle.

B.3.12 Data Integration System

Paul has eight years experience in the software industry. He is a certified in Java and Websphere, and has also worked as software architect.

The system described aggregates data from various external data sources and make them available to a number of internal applications in a large transportation company. The system was highly asynchronous and had two databases—an active and historical that kept one year's worth of data. The system dealt with over two million transactions in a day, including services to internal applications and batch processes.

The system was built by a CMMI level five company. Roles within the company were well defined. Business requirements and the main architecture were defined abroad. Paul's team was responsible by coding, deployment and maintenance. All code was revised and had to go through the formal validation of the architects. Functional and load tests were performed by a separate team. Non-functional requirements were well defined, including the expected number of concurrent requests, availability and response time. They constantly monitored the system's CPU, bandwidth, memory, queues sizes, services availability and response time. Paul describes a scalability problem triggered by external systems errors and unexpected circumstances.

In addition to provide services to other applications within the company, the system performed batch processing. These processes included tasks such as copying data from the active database to the historical one, updating the active database with external data, and removing obsolete data. They were very heavy processes that consumed a lot of the machine's resources. In order to guarantee that the system could continue to provide services to other applications, batches were scheduled to run in strict time frames, so they would not overlap. The CPU was being heavily used almost 24 hours a day. Sometimes, however, batches would overlap, drastically increasing the number of transactions the system had to handle and causing unacceptable response times to client applications. The overlapping of batches could happen for a number of reasons. For example, if an external datasource became unavailable, the system would have to wait, delaying the execution of a batch. If a backup or maintenance process was run without previous agreement, the CPU would get overloaded causing a batch processing to take longer then expected. There were also occasions where an external system would send more messages than expected and again batches would take longer to run.

These problems were expected by the company. Their solution was to have developers constantly on call to stop the system, find a time where the CPU would be less busy and manually reschedule batches. Looking back, Paul believes the company did the best it could. Keeping developers on call was, at the time, the most economical solution.

When enquired about scalability problems in general, Paul lists three common reasons for a system not to scale: bad coding, use of the wrong architecture to the problem, insufficient resources to support the load imposed on the system. He emphasizes the importance of choosing the architecture, performing load tests and experimenting with prospective technologies. Regarding the phase of the development lifecycle, the interviewee elects requirements as the most important one, followed by design, test and implementation.

B.3.13 Online Games

Anthony has ten years programming experience. He has a PhD and two postdocs in computer science. He has also four months industrial research experience. Anthony founded a online gaming company three years prior to the interview, but has not always been full-time dedicated to it. The system was developed by four people. He is the responsible by the system's back-end.

The system is a gaming infrastructure. Each game supports two online players and an unlimited number of viewers. The system also includes an online chat, where participants can discuss and start new games on virtual rooms. The Java back-end server has a number of components that implement the logging statistics, the interaction with the database, and the logic of games and rooms. Load balance occurs before the requests arrive at the Java back-end server. In theory, it is possible to redirect different rooms to distinct Java servers, although that had not been yet necessary at the time of the interview.

The system was built without following any software development methodology. There was no requirements or design phase. Developers went straight into the implementation, building what they had in mind. The games the infrastructure would support were defined by a customer, who wanted to provide a similar service to an international gaming website. Only functional tests were performed before production. Some load testing was done after the first scalability problem became apparent.

Load was estimated by looking at the international gaming website, which had around eighty thousand users. The company was developing a national website for a very small country, so they expected no more than one thousand users. The number of users grew through advertisement and word of mouth. The system was then licensed to companies in two other countries, considerably increasing the load on the server.

Anthony admits not having thought about scalability at all. The first scalability problem was hit in production, when the system reached two hundred users. Since then, the system has faced a number of scalability problems. Causes varied from threads deadlock, to badly designed SQL in the ASP pages, to improper use of regular expressions, to badly designed database maintenance routines, to network downtime. Experienced problems were memory exhaustion in the gaming server, slowness in the chat, database crashes, sudden disconnection of users, and others—either slowing down the system or making it unavailable. Problems were fixed mainly by changing the technology, such as using NIO, Java locks and a different database. The system has also gone through two major refactorings, none of then triggered by scalability problems. Even though the system had faced a number of scalability problems before these refactorings, scalability was not a concern when re-implementing the system.

Scalability problems got developers by surprise every time. Anthony says that even now, that the system is mature, scalability problems are still happening. Developers predict they will exhaust memory again when the number of users increases further, but have decided to worry about it when the problem occurs.

Looking back, Anthony does not see how the story could have been different. Scalability problems could not have been foreseen, since he did not know the technologies well enough. Anthony also declared that he would not have asked more questions to the customers and that knowing the application domain better would not have helped. The interviewee believes systems evolve naturally. Given more development time, Anthony would do more testing.

Anthony believes many systems face scalability problems because the customer is only interested in functionality and is keen on getting new features. Scalability is not a priority until hitting the system. The interviewee also believes that, at least in the gaming sector, developers should only worry about scalability during testing, after the system has been fully implemented. He also declared that thinking about scalability in the requirements phase is too early—Scalability is about optimizing bad parts of the code, which can only be detected after the system has been fully built and tested, Anthony concludes.

B.4 Critical Evaluation

A benefit of this appendix is:

Benefit: *A number of interviews with professional developers describing real scalability problems, with reflections on the development mistakes, lessons learned and desired research directions.*

Interviews were conducted in order to gain a better understanding of the treatment of scalability in industry. While the results of such interviews were not not gathered by any scientific means, it helped us to confirm our intuitions, develop a better understanding of the nature of problems normally faced in industry, and provided us with a number of examples used throughout this thesis to motivate discussions and to show the applicability of the scalability framework to different application domains and system designs.

Appendix C

A Comparative Case Study Design

This section contains the plan for a comparative case study to assess the impact of the framework in the software development lifecycle and in the scalability of the resulting system.. The plan was designed as described by Kitchenham et al. (1995). Its structure is a follows: (1) objectives, (2) hypothesis, (3) pilot project, (4) planning, (5) evaluation, (6) outcome, (7) risks, (8) concluding remarks.

C.1 Study Design: An Comparative Study [1]

C.1.1 Objectives

The study takes the form of a coding competition, open to undergraduate and graduate students. The objective of the case is to conduct a small-scale, controlled study to assess the impact of the framework in the software development lifecycle and in the scalability of the resulting system.

C.1.2 Hypothesis

1. The adoption of the proposed framework will lead to a deeper understanding of the scalability requirements/capabilities of the system, and therefore, to the ability to better justify scalability claims. Deeper understanding and better justification for scalability claims are measured in terms of the explicit definition of scalability requirements, influencing factors, measurable characteristics of the system and expected behavior for the full range of scaling characteristics.

2. The adoption of the proposed framework for early-lifecycle scalability analysis will produce a more scalable system with respect to certain system qualities (e.g. performance, resource usage, availability). The scalability is described by the causal impact that the scaling of environmental and machine characteristics have on the satisfaction of goals associated to measurable system qualities. The more scalable system will be the one that achieves the higher overall utility over the full range of scaling characteristics.

C.1.3 Pilot Project

A pilot project should be representative of the type of project that would normally benefit from the evaluated method. Although it can be argued that any software development will benefit from an upfront scalability analysis, ideally, the project should involve large amounts of data, a high-level of concurrency or complex calculations. As the time of witting, the project had not yet been defined.

Choosing students as subjects has relevant disadvantages:

1. An appropriate project for students would be too small and goals too simplistic to contain real scalability issues.

2. The framework is likely to have a greater impact than it would on an study involving experienced programmers. Controversially, students willing to participate in a coding competition are more likely to be good programmers, who would naturally consider non-functional requirements. In this case, the framework could be perceived as having little impact.

3. Students' lack of experience may jeopardize the understanding of scalability issues and identification of relevant factors that will impact scalability of the resulting system.

[1] This case study was canceled for reasons beyond our control.

4. A case study using students as subjects could be perceived by the research community as having less credibility.

The advantages of adopting a case study with students are:

1. Students can be more easily persuaded to participate on the study (by offering a money prize) and are greater in number.

2. Students should have more time and are more likely to be open to new ideas.

3. The case study may highlight a deficiency in the computer science curriculum regarding early analysis of non-functional requirements.

C.1.4 Method of Comparison

In order to test the hypotheses, the systems constructed with the aid of the scalability framework should be compared against systems developed by two other groups: one that is oblivious of the true purpose of the study and another that is told to aim for scalability but is not given the framework. This approach will also allow the evaluation of the overhead caused by the learning and execution of the proposed framework.

The hypotheses will be tested by means of questionnaires or interviews and through metrics collected from the resulting system. Comparison will take into consideration the time to complete the system, high-level design, low-level good-practice rules, achieved utility and preferences values and the participants' understanding of the scalability requirements and capabilities of the system.

C.1.5 Planning

The study will involve up to 30 students, divided into 3 groups. Group 1 will be oblivious of the true purpose of the study. Groups 2 and 3 will be told to aim for scalability, but only group 3 will be given the scalability framework.

All groups should receive a briefing and be told about the system, rules for the competition, their right to withdraw and prize. Group 3 will also receive a full lecture on the scalability framework.

All groups will be given the same time, regardless the scalability analysis and will be entitled to (reasonable) support by e-mail.

C.1.6 Prizes

Participants will be competing within their own groups for a prize of £500 (depending on sponsorship). Groups 2 and 3 will be offered an extra £100 for the most scalable system. In reality, all groups will be given an extra £100, regardless. This is just to discourage communication between the groups.
The winners will be decided according to:

- Quality of high-level design

- Values of preferences and utility

- Correctness of concurrency handling

- Design and implementation of complex algorithm
- Clarity of coding and comments
- Quality of the report and running instructions

C.1.7 Deliverables

Groups 1 and 2 are only expected to submit one deliverable (in the end of the study):

- A (3-page maximum) description of the system including a brief overview of the system, basic instructions for compiling and running the code, a simple class diagram outlining key classes and methods (hence should not contain all the methods for all classes), a description of the implementation of the complex algorithm.

Group 3 should submit two deliverables:

- First Deliverable (half-way through the study):
  - Completed scalability analysis questionnaire.
- Second Deliverable (in the end of the study):
  - Completed scalability analysis questionnaire.
  - A (3-page maximum) description of the system including a brief overview of the system, basic instructions for compiling and running the code, a simple class diagram outlining key classes and methods (hence should not contain all the methods for all classes), a description of the implementation of the complex algorithm.

Once the final deliverable has been submitted, all the participant will receive a questionnaire about the study. The questionnaire will contain scalability related questions. The answers to the questionnaire will not influence the decision on the winner. As the time of writing the questionnaires have not been defined.

C.1.8 Evaluation

Systems will be analyzed and compared according to the following aspects:

- Does the high-level design of the system indicate a concern with scalability?
- What values of preferences and utilities have been achieved for the full range and distribution of scaling characteristics for groups 2 and 3?
- Can the participant clearly express the scalability requirements and capabilities of the system?
- How well justified is the systems' scalability?
- Have the participants taken into consideration scalability concerns? How early in the lifecycle?
- Have the participants adopted low-level good-practice rules to ensure performance and scalability (e.g. in the choice of data structures and algorithms)?

Hypothesis 1 should be validated by the means of a questionnaire to assess what scalability concerns the students had in mind when building the system and how they justify their scalability claims. It is expected that the group doing the scalability analysis will have thought more thoroughly about the issues and will be able to more precisely justify claims. Additionally, the questionnaire can gather their opinion about the ease of use and relevance of the framework, as well as identify any other effect not previously envisaged.

Hypothesis 2 should be validated by comparing the systems from both groups in relation to their ability to meet requirements as characteristics of the system scale. In addition, to meet requirements, the systems should be compared according to the value of its preferences and utility, as defined in the framework.

C.1.9 Expected Outcome

The described case study, by itself, is not sufficient to validate the framework, as is not a typical IT system that would face scalability concerns and it is an one-off study. Nevertheless, it is expected to contribute to the research as following:

1. Validation of both hypotheses as stated above.
2. A better-formalized and more detailed process for scalability analysis.
3. Insights on the evolution of the scalability analysis as the software development progresses.
4. The validation of the framework on a different context from the first case study.
5. Feedback from students on the ease of use, benefits and weak points of the framework.
6. Possibly, a practitioner paper showing the benefits of teaching scalability analysis at university.
7. Hopefully, a better understanding from the students of the importance of early-lifecycle analysis of system qualities.

C.1.10 Risks

A case study with students has a few risks that should be considered. Main risks are:

1. Students may not cooperate. Students may find the proposed framework too complex/time consuming.
    - This risk can be mitigated by organizing the case study in the form of a cash-incentive competition.
2. Students project might be too small to show any advantage in adopting the process.
    - Hypothetical scalability requirements will be enforced.
3. Case study with students may not be perceived as valuable by the research community. It may be too time-consuming and with a limited value.

- Reviewers for conferences are actually encouraged to not disqualify a case study only because it uses students as subjects**?**.

4. We may simply conclude that early-lifecycle analysis of non-functional requirements will produce better quality software. This is no news to anyone.

   - That might well be the case. Although, even if the study is not successful, it should help to better structure the framework.

5. In order to be comparable, we might have to tell upfront which system qualities are being measured. That might mean that we will not be able to prove that the framework helps developers to recognize potential scalability issues.

   - That is probably the case. Although if they can still produce more scalable systems, the framework has been useful.

6. Not telling the control group that the research is about scalability, might be perceived as cheating, as they obviously will not be concentrating on the problem.

   - Not telling the control group the actual objective of the research is not really a problem, because in the real-world developers and are normally more concerned with the functional side of the system and that is why non-functional concerns like scalability end up being overlooked. The participants will be divided in three groups to overcome this problem:

     i. A control group that should have the main focus on the functional requirements of the system,

     ii. An unaided scalability group that are explicitly told scalability should be a concern,

     iii. A group using the scalability framework.

     All of them should receive the same requirements document, but having different emphasis on the briefing.

7. Control group will realize the actual objective of the framework when it is asked to answer the scalability questionnaire. They might then pretend they thought about scalability more than they actually have.

   - The scalability questionnaire should only be received after submitting the final code. It should be online, not letting students to go back to previous questions. Questions should be careful planned not to give away what is coming next. Also, there should be a time limit for answering the questionnaire, as we want to get the first reactions. Failure to do so will invalidate the participants entry to the prize.

8. Participants may look us up on the Internet and find out what the research is actually about.

   - Hide true identity of people behind the research. Control group should be given the briefing by someone not related to the research. Submissions should be online and a specific e-mail account should be set up for questions.

9. Because groups are not competing with each other, they might interact, jeopardizing the study.

    - An extra £100 should be offered to the best of all group winners. In reality, all winners would receive the extra £100, as the resulting systems cannot be fairly compared.

10. Learning curve of the framework might mean that group performing the analysis will be in disadvantage regarding time.

    - This is a question that should be included in the questionnaire. If we conclude the framework is too high overhead, then we will have to re-think it, anyway. Furthermore, because the groups are not competing between themselves, there is no problems about fairness.

11. We might not get enough volunteers.

    - Participants should be informed that the study will only go ahead if a minimum number of volunteers is achieved.

C.1.11 Concluding Remarks

This document has described a prospective case study based on a coding competition for students. The main points are:

- The adoption of the scalability framework for early-lifecycle analysis should produce more scalable systems and provide a deeper understanding of the scalability requirements/capabilities of software systems.

- The case study will involve up to 30 students and 3 distinct groups. The hypotheses will be tested by means of questionnaires and through metrics collected from the resulting system.

- The case study, by itself, is not sufficient to validate the framework, as it is not a typical IT system that would face scalability concerns and it is an one-off study. This problem is addressed by Case Study 2, who uses a real-world system and experienced developers. This study can, however, contribute to the research in many ways, such as a better formalized framework, a greater understanding of the overhead and benefits of scalability analysis and a comparison of the influence of the framework in the scalability of the final system.

Bibliography

Akao, Y. (1990), *Quality Function Deployment QFD: Integrating Customer Requirements into Product Design*, Productivity Press.

Alazzawi, L. K., Elkateeb, A. M., Ramesh, A. and Aljuhar, W. (2008), Scalability Analysis for Wireless Sensor Networks Routing Protocols, *in AINA Workshop 2008: Proceedings of the 22nd International Conference on Advanced Information Networking and Applications*, IEEE Computer Society, pp.139–144.

Alspaugh, T. (2007), Informal Conversation.

Anderson, K. M. (1999a), Data scalability in open hypermedia systems, *in HYPERTEXT'99: Proceedings of the tenth ACM Conference on Hypertext and hypermedia: returning to our diverse roots*, ACM, New York, NY, USA, pp.27–36.

Anderson, K. M. (1999b), Supporting industrial hyperwebs: lessons in scalability, *in ICSE'99: Proceedings of the 21st international conference on Software engineering*, IEEE Computer Society Press, Los Alamitos, CA, USA, pp.573–582.

Arlitt, M., Krishnamurthy, D. and Rolia, J. (2001), Characterizing the scalability of a large Web-based shopping system, *ACM Transactions on Internet Technology (TOIT)* **1**(1), 44–69.

Avesani, P., Bazzanella, C., Perini, A. and Susi, A. (2005), Facing Scalability Issues in Requirements Prioritization with Machine Learning Techniques, *in RE'05: Proceedings of the 13th IEEE International Conference on Requirements Engineering*, IEEE Computer Society, Washington, DC, USA, pp.297–306.

Avritzer, A., Kondek, J., Liu, D. and Weyuker, E. J. (2002), Software Performance Testing Based on Workload Characterization, *in Software Performance Testing Based on Workload Characterization*, ACM Press, pp.17–24.

Bahsoon, R. and Emmerich, W. (2008), An Economics-Driven Approach for Valuing Scalability in Distributed Architectures, *in WICSA'08: Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*, IEEE Computer Society, Washington, DC, USA, pp.9–18.

Barra, M., Cattaneo, G., Petrillo, U. F. and Scarano, V. (2001), JSEB (Java Scalable sErvices Builder): Scalable Systems for Clusters of Workstations, *in ISCC'01: Proceedings of the Sixth IEEE Symposium on Computers and Communications*, IEEE Computer Society, Washington, DC, USA, p.80.

Barroso, L. A. (2005), The Price of Performance, *ACM Queue: Multiprocessors* **3**(7).

Barroso, L. A., Dean, J. and Hölzle, U. (2003), Web Search for a Planet: The Google Cluster Architecture, *IEEE Micro* **23**(2), 22–28.

Beck, K. and Andres, C. (2004), *Extreme Programming Explained: Embrace Change (2nd Edition)*, Addison-Wesley Professional.

Bengtsson, P. and Bosch, J. (1998), Scenario-Based Software Architecture Reengineering, *in ICSR'98: Proceedings of the 5th International Conference on Software Reuse*, IEEE Computer Society, Washington, DC, USA, p.308.

Bennani, M. N. and Menasce, D. A. (2005), Resource Allocation for Autonomic Data Centers using Analytic Performance Models, *in ICAC'05: Proceedings of the Second International Conference on Automatic Computing*, IEEE Computer Society, Washington, DC, USA, pp.229–240.

Bertolino, A. and Mirandola, R. (2004), Software Performance Engineering of Component-Based Systems, *in WOSP'04: Proceedings of the 4th international workshop on Software and performance*, ACM Press, New York, USA, pp.238–242.

Bhushan, B. and Patel, A. (1998), Requirements and the concept of cooperative system management, *International Journal of Network Management* **8**(3), 139–158.

Bivens, A., Gupta, R., McLean, I., Szymanski, B. and White, J. (2004), Scalability and performance of an agent-based network management middleware, *International Journal of Network Management* **14**(2), 131–146.

Blockeel, H. and Sebag, M. (2003), Scalability and efficiency in multi-relational data mining, *SIGKDD Explorations Newsletter* **5**(1), 17–30.

Boehm, B. (2006), A view of 20th and 21st century software engineering, *in ICSE'06: Proceedings of the 28th international conference on Software engineering*, ACM, New York, NY, USA, pp.12–29.

Boehm, B. W. (1976), Software engineering, pp.1226–1241.

Boehm, B. W., Brown, J. R. and Lipow, M. (1976), Quantitative evaluation of software quality, *in ICSE'76: Proceedings of the 2nd international conference on Software engineering*, IEEE Computer Society Press, Los Alamitos, CA, USA, pp.592–605.

Bondi, A. B. (2000), Characteristics of Scalability and Their Impact on Performance, *in WOSP'00: Proceedings of the* 2^{nd} *International Workshop on Software and Performance*, ACM Press, pp.195–203.

Brataas, G. and Hughes, P. (2004), Exploring Architectural Scalability, *in WOSP'04: Proceedings of the 4^{th} International Workshop on Software and Performance*, ACM Press, pp.125–129.

Buehrer, G. and Chellapilla, K. (2008), A scalable pattern mining approach to web graph compression with communities, *in WSDM'08: Proceedings of the international conference on Web search and web data mining*, ACM, New York, NY, USA, pp.95–106.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. (1996), *Pattern-oriented software architecture: a system of patterns*, John Wiley & Sons, Inc., New York, NY, USA.

Callow, M., Beardow, P. and Brittain, D. (2007), Big games, small screens, *Queue* **5**(7), 40–50.

Caporuscio, M., Marco, A. D. and Inverardi, P. (2005), Run-time Performance Management of the Siena Publish/Subscribe Middleware, *in WOSP'05: Proceedings of the 5th international workshop on Software and performance*, ACM Press, New York, USA, pp.65–74.

Capra, L., Emmerich, W. and Mascolo, C. (2002), A micro-economic approach to conflict resolution in mobile computing, *SIGSOFT Software Engineering Notes* **27**(6), 31–40.

Cecchet, E., Marguerite, J. and Zwaenepoel, W. (2002), Performance and scalability of EJB applications, *SIGPLAN Notices* **37**(11), 246–261.

Chen, Y. and Sun, X.-H. (2006), STAS: A Scalability Testing and Analysis System, *Proceedings of 2006 IEEE International Conference on Cluster Computing* pp.1–10.

Cheng, B. H. C. and Atlee, J. M. (2007), Research Directions in Requirements Engineering, *in FOSE'07: Prooceedings of 2007 Future of Software Engineering*, IEEE Computer Society, Washington, DC, USA, pp.285–303.

Cheng, S.-W., Garlan, D. and Schmerl, B. (2006), Architecture-based self-adaptation in the presence of multiple objectives, *in SEAMS'06: Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems*, ACM, New York, NY, USA, pp.2–8.

Chiu, W. (2001), Design for Scalability — An Update. High Volume Web Sites, Software Group (AIM Division). Accessed from http://www.ibm.com/developerworks/websphere/library/techarticles/hipods/scalability.html on 13 March 2009.

Chung, L., Nixon, B. A., Yu, E. and Mylopoulos, J. (1999), *Non-Functional Requirements in Software Engineering*, Springer.

Clarke, E. M., Enders, R., Filkorn, T. and Jha, S. (1996), Exploiting symmetry in temporal logic model checking, *Formal Methods in System Design* **9**(1-2), 77–104.

Clements, P., Kazman, R. and Klein, M. (2002), *Evaluating Software Architectures: Methods and Case Studies*, Addison-Wesley Professional.

Coarfa, C., Mellor-Crummey, J., Froyd, N. and Dotsenko, Y. (2007), Scalability analysis of SPMD codes using expectations, *in ICS'07: Proceedings of the 21st annual international conference on Supercomputing*, ACM, New York, NY, USA, pp.13–22.

ComputerUser (2008), Scalability Definition. Accessed from http://www.computeruser.com/resources/dictionary/noframes/nf.definition.html?bG9va3VwPTQ5ODY= on 18 August 2008.

Courtois, P.-J. and Parnas, D. L. (1993), Documentation for safety critical software, *in ICSE'93: Proceedings of the 15th international conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos, CA, USA, pp.315–323.

D'Antonio, S., Esposito, M., Romano, S. P. and Ventre, G. (2004), Assessing the scalability of component-based frameworks: the CADENUS case study, *SIGMETRICS Perform. Eval. Rev.* **32**(3), 34–43.

Darimont, R. and van Lamsweerde, A. (1996), Formal refinement patterns for goal-driven requirements elaboration, *in SIGSOFT'96: Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, ACM, New York, NY, USA, pp.179–190.

Deters, R. (2001), Scalability and information agents, *SIGAPP Applied Computing Review* **9**(3), 13–20.

DoD (1995), Department of Defense Modeling and Simulation (M&S) Master Plan. DoD reference 5000.59.

Dromey, R. G. (1996), Concerning the Chimera [software quality], *IEEE Software* (1), 33–43.

Duboc, L., Letier, E., Rosenblum, D. and Wicks, T. (2008), Case Study in Eliciting Scalability Requirements, *in RE'08: Proceedings of the 2008 16th IEEE International Requirements Engineering Conference*, Barcelona, Spain.

Duboc, L., Rosenblum, D. and Wicks, T. (2007), A Framework for Characterization and Analysis of Software system Scalability, *in ESEC-FSE'07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ACM, New York, NY, USA, pp.375–384.

Duboc, L., Rosenblum, D. S. and Wicks, T. (2006), A framework for modelling and analysis of software systems scalability, *in ICSE '06: Proceedings of the 28th international conference on Software engineering. Doctoral Symposium.*, ACM, New York, NY, USA, pp.949–952.

Egea-Lopez, E., Vales-Alonso, J., Martinez-Sala, A., Pavon-Mario, P. and Garcia-Haro, J. (2006), Simulation scalability issues in wireless sensor networks, *IEEE Communications Magazine* **44**(7), 64–73.

Elssamadisy, A. and Schalliol, G. (2002), Recognizing and responding to "bad smells" in extreme programming, *in ICSE'02: Proceedings of the 24th International Conference on Software Engineering*, ACM, New York, NY, USA, pp.617–622.

EMC^2 (2008), Documentum Platform. Accessed from http://uk.emc.com/ on 5 August 2008.

Emerson, E. A. and Sistla, A. P. (1997), Utilizing symmetry when model-checking under fairness assumptions: an automata-theoretic approach, *ACM Transactions on Programming Languages and Systems (TOPLAS)* **19**(4), 617–638.

Eschenauer, H., Koski, J. and Osyczka, A. (1990), *Multicriteria Design Optimization: Procedures and Applications*, Springer-Verlag.

Farlex (2008), The Free Dictionary: Scalable. Accessed from http://computing-dictionary.thefreedictionary.com/scalable on 18 August 2008.

Feather, M., Cornford, S., Dunphy, J. and Hicks, K. (2002), A Quantitative Risk Model for Early Lifecycle Decision Making, *in IDPT'02: Proceedings of the Conference on Integrated Design and Process Technology*.

Feather, M. S. (1987), Language support for the specification and development of composite systems, *ACM Transactions on Programming Languages and Systems (TOPLAS)* **9**(2), 198–234.

Fenton, N. E. and Pfleeger, S. L. (1996), *Software Metrics: A Rigorous and Practical Approach*, International Thomson Computer Press, Boston, MA, USA.

Ficici, S. G. and Pfeffer, A. (2008), Simultaneously modeling humans' preferences and their beliefs about others' preferences, *in AAMAS'08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, pp.323–330.

Fielding, R. T. and Taylor, R. N. (2000), Principled design of the modern Web architecture, *in ICSE'00: Proceedings of the 22nd international conference on Software engineering*, ACM, New York, NY, USA, pp.407–416.

Fielding, R. T. and Taylor, R. N. (2002), Principled design of the modern Web architecture, *ACM Transactions on Internet Technology (TOIT)* **2**(2), 115–150.

Finkelstein, A. and Dowell, J. (1996), A comedy of errors: the London Ambulance Service case study, *in IWSSD'96: Proceedings of the 8th International Workshop on Software Specification and Design*, IEEE Computer Society, Washington, DC, USA, p.2.

Gabriel, R. P., Northrop, L., Schmidt, D. C. and Sullivan, K. (2006), Ultra-large-scale systems, *in OOPSLA'06: Prooceedings of Dynamic Languages Symposium archive Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, ACM, New York, NY, USA, pp.632–634.

Ghemawat, S., Gobioff, H. and Leung, S.-T. (2003), The Google file system, *SOSP'03: Proceedings of the nineteenth ACM symposium on Operating systems principles* **37**(5), 29–43.

Giorgini, P., Mylopoulos, J., Nicchiarelli, E. and Sebastiani, R. (2002), Reasoning with Goal Models, *in ER'02: Proceedings of the 21st International Conference on Conceptual Modeling*, Springer-Verlag, London, UK, pp.167–181.

Goldsmith, S. F., Aiken, A. S. and Wilkerson, D. S. (2007), Measuring empirical computational complexity, *in ESEC-FSE'07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ACM, New York, NY, USA, pp.395–404.

Grama, A. Y., Gupta, A. and Kumar, V. (1993), Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures, *IEEE Parallel & Distributed Technology: Systems & Technology* **1**(3), 12–21.

Grumberg, E. M. C. O. and Peled, D. A. (1999), *Model Checking*, The MIT PRess.

Gustafson, J. L. (1995), Reevaluating Amdahl's law, *Multiprocessor performance measurement and evaluation* pp.92–93.

Gustavson, D. B. (1994), The Many Dimensions of Scalability, *in Proceedings of the* 39^{th} *IEEE Computer Society International Computer Conference*, pp.60–63.

Handley, M., Perkins, C. and Whelan, E. (2000), Session Announcement Protocol, RFC 2974, Network Working Group, Internet Engineering Task Force. Accessed from http://www.faqs.org/ftp/rfc/pdf/rfc2974.txt.pdf on 3 April 2006.

Heindl, M. and Biffl, S. (2006), Risk management with enhanced tracing of requirements rationale in highly distributed projects, *in GSD'06: Proceedings of the 2006 international workshop on Global software development for the practitioner*, ACM, New York, NY, USA, pp.20–26.

Hill, M. D. (1990), What is Scalability?, *ACM SIGARCH Computer Architecture News* **18**(4), 18–21.

Hoeben, F. (2000), Using UML models for performance calculation, *in WOSP'00: Proceedings of the 2nd international workshop on Software and performance*, ACM Press, pp.77–82.

Hopcroft, J. E., Motwani, R. and Ullman, J. D. (2006), *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Hopkins, R. P., Smith, M. J. and King, P. J. B. (2002), Two Approaches to Integrate UML and Performance Models, *in WOSP'03: Proceedings of the 3rd international workshop on Software and performance*, ACM Press, pp.91–92.

Imafouo, A. (2005), An Scalability Survey, *in ECDL 2005: Proceedings of the 9th European Conference on Research and Advanced Technology for Digital Libraries — Doctoral Consortium*, Vienna, Austria.

In, H., Boehm, B., Rodgers, T. and Deutsch, M. (2001), Applying WinWin to quality requirements: a case study, *in ICSE'01: Proceedings of the 23rd International Conference on Software Engineering*, IEEE Computer Society, Washington, DC, USA, pp.555–564.

Industry Interviews (2007), Examples of Scalability Problems in Industry. Private communication.

Ip, C. N. and Dill, D. L. (1996), Better verification through symmetry, *Formal Methods in System Design* **9**(1-2), 41–75.

ISO 9126 (1991), Software Product Evaluation - Quality characteristics and guidelines for their use.

Jackson, M. (2001), *Problem frames: analyzing and structuring software development problems*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Jacobs, D. (2005), Enterprise software as service, *Queue* **3**(6), 36–42.

Ji, Z., Zhou, J., Takai, M. and Bagrodia, R. (2006), Improving scalability of wireless network simulation with bounded inaccuracies, *ACM Transactions on Modeling and Computer Simulation (TOMACS)* **16**(4), 329–356.

Jiang, X., Safaei, F. and Boustead, P. (2005), Latency and scalability: a survey of issues and techniques for supporting networked games, *in Proceedings of the 13th IEEE International Conference on Networks, 2005. Jointly held with the 2005 IEEE 7th Malaysia International Conference on Communication*.

Jogalekar, P. and Woodside, M. (1998), Evaluating the Scalability of Distributed Systems, *in HICSS'98: Proceedings of the* 31^{st} *Annual Hawaii International Conference on System Sciences, Volume 7*, IEEE Computer Society, Washington, DC, USA, pp.524–531.

Jogalekar, P. and Woodside, M. (2000), Evaluating the Scalability of Distributed Systems, *IEEE Transactions on Parallel and Distributed Systems* **11**(6), 589–603.

Jogalekar, P. P. and Woodside, C. M. (1997), A Scalability Metric for Distributed Computing Applications in Telecommunications, *in ITC'97: Proceeding of the* 15^{th} *International Teletraffic Congress*, Vol. 2a, pp.101–110.

Jones, T., Dawson, S., Neely, R., Tuel, W., Brenner, L., Fier, J., Blackmore, R., Caffrey, P., Maskell, B., Tomlinson, P. and Roberts, M. (2003), Improving the Scalability of Parallel Jobs by adding Parallel Awareness to the Operating System, *in SC'03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, IEEE Computer Society, Washington, DC, USA, p.10.

Kalinov, A. (2004), Scalability Analysis of Matrix-Matrix Multiplication on Heterogeneous Clusters, *in ISPDC'04: Proceedings of the Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, IEEE Computer Society, Washington, DC, USA, pp.303–309.

Kang, S., Lee, J., Jang, H., Lee, H., Lee, Y., Park, S., Park, T. and Song, J. (2008), SeeMon: scalable and energy-efficient context monitoring framework for sensor-rich mobile environments, *in MobiSys'08: Proceeding of the 6th international conference on Mobile systems, applications, and services*, ACM, New York, NY, USA, pp.267–280.

Kazman, R., Abowd, G., Bass, L. and Clements, P. (1996), Scenario-Based Analysis of Software Architecture, *IEEE Software* **13**(6), 47–55.

Kazman, R., Asundi, J. and Klein, M. (2001), Quantifying the costs and benefits of architectural decisions, *in ICSE'01: Proceedings of the 23rd International Conference on Software Engineering*, IEEE Computer Society, Washington, DC, USA, pp.297–306.

Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H. and Carriere, J. (1998), The architecture tradeoff analysis method, *in ICECCS'98: Proceedings. Fourth IEEE International Conference on Engineering of Complex Computer Systems*, pp.68–78.

Keller, S. E., Kahn, L. G. and Panara, R. B. (1990), Specifying Software Quality Requirements with Metrics, *in Tutorial: System and Software Requirements Engineering*, IEEE Computer Society Press, pp.145–163.

Kim, K.-H. and Ellis, C. A. (2001), Workflow performance and scalability analysis using the layered queuing modeling methodology, *in GROUP'01: Proceedings of the 2001 International ACM SIGGROUP Conference on Supporting Group Work*, ACM, New York, NY, USA, pp.135–143.

Kitchenham, B., Pickard, L. and Pfleeger, S. L. (1995), Case Studies for Method and Tool Evaluation, *IEEE Software* **12**(4), 52–62.

Klein, M. H., Kazman, R., Bass, L. J., Carrière, S. J., Barbacci, M. and Lipson, H. F. (1999), Attribute-Based Architecture Styles, *in WICSA1: Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, Kluwer, B.V., Deventer, The Netherlands, The Netherlands, pp.225–244.

Kong, J., Bridgewater, J. and Roychowdhury, V. (2006), A General Framework for Scalability and Performance Analysis of DHT Routing Systems, *DSN'06: International Conference on Dependable Systems and Networks* pp.343–354.

Kosch, T., Adler, C. J., Eichler, S., Schroth, C. and Strassberger, M. (2006), The scalability problem of vehicular ad hoc networks and how to solve it, **13**, 22–28.

Koymans, R. (1992), *Specifying Message Passing and Time-Critical Systems with Temporal Logic*, Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Kruchten, P. (1999), *The Rational Unified Process: an introduction*, Addison-Wesley Longman Publishing Co., Boston, MA, USA.

Kumar, V. and Gupta, A. (1994), Analyzing Scalability of Parallel Algorithms and Architectures, *Journal of Parallel and Distributed Computing* **22**(3), 379–391.

Kwok, M.-M. and Wong, S. M.-J. W. (2008), Scalability Analysis of the Hierarchical Architecture for Distributed Virtual Environments, *IEEE Transactions on Parallel and Distributed Systems* **19**(3), 408–417.

Laitinen, M. (2000), Scaling down is hard to do [software management], *IEEE Software* **17**(5), 78–80.

Laitinen, M., Fayad, M. E. and Ward, R. P. (2000), Thinking Objectively: The problem with Scalability, *Communications of the ACM* **43**(9), 105–107.

Lamparter, S., Ankolekar, A., Studer, R. and Grimm, S. (2007), Preference-based selection of highly configurable web services, *in WWW'07: Proceedings of the 16th international conference on World Wide Web*, ACM, New York, NY, USA, pp.1013–1022.

Law, D. R. (1998), Scalable Means More than More: A Unifying Definition of Simulation Scalability, *in WSC'98: Proceedings of the 30th conference on Winter simulation*, IEEE Computer Society Press, Los Alamitos, CA, USA, pp.781–788.

Lee, C. B. and Snavely, A. E. (2007), Precise and realistic utility functions for user-centric performance analysis of schedulers, *in HPDC'07: Proceedings of the 16th international symposium on High performance distributed computing*, ACM, New York, NY, USA, pp.107–116.

Letier, E. (2001), Reasoning about Agents in Goal-Oriented Requirements Engineering. Phd Thesis, Universit Catholique de Louvain, Dpartement d'Ingnierie Informatique, Louvain-la-Neuve, Belgium.

Letier, E. and van Lamsweerde, A. (2002), Agent-based tactics for goal-oriented requirements elaboration, *in ICSE'02: Proceedings of the 24th International Conference on Software Engineering*, ACM, New York, NY, USA, pp.83–93.

Letier, E. and van Lamsweerde, A. (2004), Reasoning about partial goal satisfaction for requirements and design engineering, *in FSE'04: Proceedings of the 12th international symposium on Foundations of software engineering*, ACM, New York, NY, USA, pp.53–62.

Li, Q., Xu, M., Xu, K. and Wu, J. (2008), Evaluating Service Scalability of Network Architectures, *ICN'08: Proceedings of the Seventh International Conference on Networking* **0**, 434–438.

Liu, Y., Gorton, I., Liu, A. and Chen, S. (2002), Evaluating the Scalability of Enterprise JavaBeans Technology, *in APSEC'02: Proceedings of the Ninth Asia-Pacific Software Engineering Conference*, IEEE Computer Society, Washington, DC, USA, p.74.

Liu, Y. and Gorton, I. (2004), Accuracy of Performance Prediction for EJB Applications: A Statistical Analysis., *in SEM'04: Proceedings of the 4^{th} Workshop on Software Engineering and Middleware*, pp.185–198.

Llado;, C. M. and Harrison, P. G. (2000), Performance Evaluation of an Enterprise JavaBean Server Implementation, *in WOSP'00: Proceedings of the 2nd international workshop on Software and performance*, ACM Press, New York,USA, pp.180–188.

Lu, N. and Bigham, J. (2006), An optimal bandwidth adaptation algorithm for multi-class traffic in wireless networks, *in QShine'06: Proceedings of the 3rd international conference on Quality of service in heterogeneous wired/wireless networks*, ACM, New York, NY, USA, p.55.

Luke, E. A. (1993), Defining and Measuring Scalability, *in Proc. Scalable Parallel Libraries Conference*, IEEE Press, pp.183–186.

Lutu, P. E. N. (2002), An integrated approach for scaling up classification and prediction algorithms for data mining, *in SAICSIT'02: Proceedings of the 2002 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology*, South African Institute for Computer Scientists and Information Technologists, Republic of South Africa, pp.110–117.

Lyu, M. R. (ed.) (1996), *Handbook of software reliability engineering*, McGraw-Hill, Inc., Hightstown, NJ, USA.

Marbukh, V. (2007), Utility maximization for resolving throughput/reliability trade-offs in an unreliable network with multipath routing, *in ValueTools'07: Proceedings of the 2nd international conference on Performance evaluation methodologies and tools*, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, pp.1–10.

Masticola, S., Bondi, A. B. and Hettish, M. (2005), Model-Based Scalability Estimation in Inception-Phase Software Architecture., *in MODELS'05: Prooceedings of the 8th International Conference on Model Driven Engineering Languages and Systems*, pp.355–366.

McCall, J., Richards, P. and Walters, G. (1977), *Factors in Software Quality, Vols 1-3*, NTIS AD-A049-014, AD-A049-015, AD-A049-055.

Menascé, D. (2003), Scaling Web Sites Through Caching, *IEEE Internet Computing* **7**(4), 86–89.

Menasce, D. A. and Almeida, V. (2001), *Capacity Planning for Web Services: metrics, models, and methods*, Prentice Hall PTR, Upper Saddle River, NJ, USA.

Messerschmitt, D. (1995), The convergence of telecommunications and computing: What are the implications today?, *in CWS'95: Proceedings of the 27th Conference on Winter Simulation*, IEEE Computer Society Press, pp.1280 – 1287.

Meter, R. V. and Oskin, M. (2006), Architectural implications of quantum computing technologies, *Journal on Emerging Technologies in Computing Systems (JETC)* **2**(1), 31–63.

Microsoft (2005), Designing Distrubuted Systems with .Net. MSDN library. Accessed from http://msdn.microsofte.com/library/default.asp?url=/library/en-us/vsent7/html/vxconwhtisscalability.asp on 3 July 2005.

Monch, C. (2002), On the assessment of scalability of digital libraries, *in Proceedings of the Fourth DELOS Workshop. Evaluation of Digital Libraries: Testbeds, Measurements, and Metrics*.

Mos, A. and Murphy, J. (2004), COMPAS: Adaptive Performance Monitoring of Component-Based Systems, *in WCOP'04: Proceedings of the* 9^{th} *International Workshop on Component Oriented Programming*, IEEE Computer Society, Washington, DC, USA.

Müller, J. and Gorlatch, S. (2006), Rokkatan: scaling an RTS game design to the massively multiplayer realm, *Comput. Entertain.* **4**(3), 11.

Munsterman, T. P. (2008), Exploring Software Scalability. Thesis submitted for the degree of Master of Science at the University of Twene.

Musa, J. D. (1993), Operational Profiles in Software-Reliability Engineering, *IEEE Software* **10**(2), 14–32.

Nadeau, T. P. and Teorey, T. J. (2002), Achieving scalability in OLAP materialized view selection, *in DOLAP'02: Proceedings of the 5th ACM international workshop on Data Warehousing and OLAP*, ACM, New York, NY, USA, pp.28–34.

Noelle, M., Pantano, M. and Sun, X.-H. (1998), Communication Overhead: Prediction and its Influence on Scalability, *in PDPTA'98: Proceesings of the International Conference on Parallel and Distributed Processing Techniques and Applications*.

North, C. (2006), The Perceptual Scalability of Visualization, *IEEE Transactions on Visualization and Computer Graphics* **12**(5), 837–844. Student Member-Beth Yost.

Odersky, M., Spoon, L. and Venners, B. (2008), Scala: A Scalable Language. Accessed from http://www.artima.com/scalazine/articles/scalable-language.html on 14 August 2008.

Page, D., Williams, P. and Boyd, D. (1993), Report of the Inquiry Into the London Ambulance Service, Technical Report, The Communications Directorate, South West Thames Regional Authority. Accessed from http://hsn.lond-amb.sthames.nhs.uk/http.dir/service/organisation/features/info.html.

Papavassiliou, S., Xu, S., Orlik, P., Snyder, M. and Sass, P. (2002), Scalability in global mobile information systems (GloMo): issues, evaluation methodology and experiences, *Wireless Networks* **8**(6), 637–648.

Pastor, L. and Bosque, J. L. (2001), An efficiency and scalability model for heterogeneous clusters., *in CLUSTER'01: Proceedings of the 3rd IEEE International Conference on Cluster Computing*, IEEE Computer Society, Washington, DC, USA, p.427.

Petriu, D., Amer, H., Majumdar, S. and Abdull-Fatah, I. (2000), Using Analytic Models Predicting Middleware Performance, *in WOSP'00: Proceedings of the 2nd international workshop on Software and performance*, ACM Press, NYNew York, USA, pp.189–194.

Potts, C. (1995), Using schematic scenarios to understand user needs, *in DIS'95: Proceedings of the 1st conference on Designing interactive systems*, ACM, New York, NY, USA, pp.247–256.

Quinn, M. (1994), *Parallel Computing: Theory and Practice*, 2nd edition, McGraw-Hill.

Rana, O. F. and Stout, K. (2000), What is scalability in multi-agent systems?, *in AGENTS'00: Proceedings of the fourth international conference on Autonomous agents*, ACM, New York, NY, USA, pp.56–63.

Rao, V. N. and Kumar, V. (1987), Parallel depth first search. Part I. implementation, *International Journal of Parallel Programming* **16**(6), 479–499.

Reiss, S. P. (2008), Controlled dynamic performance analysis, *in WOSP'08: Proceedings of the 7th international workshop on Software and performance*, ACM, New York, NY, USA, pp.43–54.

Rivest, R. L. and Leiserson, C. E. (1990), *Introduction to Algorithms*, McGraw-Hill, Inc., New York, NY, USA.

Robertson, S. and Robertson, J. (1999), *Mastering the requirements process*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

Robinson, W. N. (1990), Negotiation behavior during requirements specification, *in ICSE'90: Proceedings of the 12th international conference on Software engineering*, IEEE Computer Society Press, Los Alamitos, CA, USA, pp.268–276.

Robu, V., Somefun, D. J. A. and Poutré, J. A. L. (2005), Modeling complex multi-issue negotiations using utility graphs, *in AAMAS'05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, ACM, New York, NY, USA, pp.280–287.

Ruthruff, J. R., Elbaum, S. and Rothermel, G. (2006), Experimental program analysis: a new program analysis paradigm, *in ISSTA'06: Proceedings of the 2006 international symposium on Software testing and analysis*, ACM Press, New York, USA, pp.49–60.

Saaty, T. L. (1980), *The Analytic Hierarchy Process*, McGraw-Hill, New York, NY, USA.

Sastry, K., Goldberg, D. E. and Pelikan, M. (2005), Limits of scalability of multiobjective estimation of distribution algorithms, *in Congress on Evolutionary Computation*, IEEE, pp.2217–2224.

Scholz, U. and Rouvoy, R. (2007), Divide and conquer: scalability and variability for adaptive middleware, *in ESSPE'07: International workshop on Engineering of software services for pervasive environments*, ACM, New York, NY, USA, pp.35–39.

SearchDataCenter (2006). Accessed from http://searchdatacenter.techtarget.com/sDefinition/0,,sid80_gci212940,00.html on 18 August 2008.

Shaw, M. and Garlan, D. (1996), *Software architecture: perspectives on an emerging discipline*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Shen, H., Kumar, M., Das, S. K. and Wang, Z. (2005), Energy-efficient data caching and prefetching for mobile devices based on utility, *Mobile Networks and Applications* **10**(4), 475–486.

Silva, D. D., Krieger, O., Wisniewski, R. W., Waterland, A., Tam, D. and Baumann, A. (2006), K42: an infrastructure for operating system research, *ACM SIGOPS Operating Systems Review* **40**(2), 34–42.

Silverman, R. (2000), A cost-based security analysis of symmetric and asymmetric key lengths. RSA Laboratories Bulletin 13.

Smith, C. U. and Williams, L. G. (2001), *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*, Addison-Wesley Publishing Company.

Sommerville, I. (2004), *Software Engineering (7th Edition)*, Pearson Addison Wesley.

Sorensen, S.-A. (2007), Informal Conversation.

Steen, M. V., van der Zijden, S. and Sips, H. J. (1998), COMPSAC '98: Software Engineering for Scalable Distributed Applications, *in Proceedings of the 22nd International Computer Software and Applications Conference*, pp.285–293.

Stephens, J. M. and Poess, M. (2004), MUDD: A Multi-Dimensional Data Generator, *in WOSP'04: Proceedings of the 4th international workshop on Software and performance*, ACM Press, pp.104–109.

Sun, X.-H. (2002), Scalability versus Execution Time in Scalable Systems, *Journal of Parallel and Distributed Computing* **62**, 173–192.

Sun, X.-H., Chen, Y. and Wu, M. (2005), Scalability of Heterogeneous Computing, *in ICPP'05: Proceedings of the 2005 International Conference on Parallel Processing*, IEEE Computer Society, Washington, DC, USA, pp.557–564.

Sun, X.-H. and Gustafson, J. L. (1993), Toward a better parallel performance metric, pp.23–39.

Sun, X. H. and Rover, D. T. (1994), Scalability of Parallel Algorithm-Machine Combinations, *IEEE Transactions on Parallel and Distributed Systems* **5**(6), 599–613.

TCSC (2008), IEEE Technical Committee on Scalable Computing (TCSC). Accessed from http://www.ieeetcsc.org/ on 5 August 2008.

Tepfenhart, W., Rosca, D. and Woolley, D. (2002), A product focused, layered software development framework, *in SEKE'02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*, ACM, New York, NY, USA, pp.473–475.

The 2020 Science Group (2006), Towards 2020 Science, technical report, Microsoft Corporation. Accessed from http://research.microsoft.com/towards2020science/ on 1 July 2006.

Tribastone, M. and Gilmore, S. (2008), Automatic extraction of PEPA performance models from UML activity diagrams annotated with the MARTE profile, *in WOSP'08: Proceedings of the 7th international workshop on Software and performance*, ACM, New York, NY, USA, pp.67–78.

University of Wiscousin-Madison (2006), Condor High Computing Manual Pages, version 6.7.20. http://www.cs.wisc.edu/condor/manual/v6.7/.

van Lamsweerde, A. (2001), Goal-Oriented Requirements Engineering: A Guided Tour, *in RE'01: Proceedings of the 5^{th} IEEE International Symposium on Requirements Engineering*, IEEE Computer Society, Washington, DC, USA, p.249.

van Lamsweerde, A. (2008), *Systematic Requirements Engineering - From System Goals to UML Models to Software Specifications*, John Wiley & Sons.

van Lamsweerde, A. and Letier, E. (2000), Handling Obstacles in Goal-Oriented Requirements Engineering, *IEEE Transactions on Software Engineering* **26**(10), 978–1005.

van Lamsweerde, A., Letier, E. and Darimont, R. (1998), Managing Conflicts in Goal-Driven Requirements Engineering, *IEEE Transactions on Software Engineering* **24**(11), 908–926.

Vanier, M. (2001), Scalable computer programming languages. Accessed from http://www.cs.caltech.edu/rogramming_languages.html on 14 August 2008.

Varian, H. R. (2003), *Intermediate Microeconomics: A Modern Approach*, 6th edition, W. W. Norton.

Verdickt, T., Dhoedt, B. and Gielen, F. (2004), Incorporating SPE into MDA: Including Middleware Performance Details Into System Models, *ACM SIGSOFT Software Engineering Notes* **29**(1), 120–124.

Vetter, J. S. and McCracken, M. O. (2001), Statistical scalability analysis of communication operations in distributed applications, *in PPoPP'01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, ACM, New York, NY, USA, pp.123–132.

von Behren, R., Condit, J., Zhou, F., Necula, G. C. and Brewer, E. (2003), Capriccio: scalable threads for internet services, *in SOSP'03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, ACM, New York, NY, USA, pp.268–281.

Webopedia (2006), What is Scalable? Accessed from http://www.webopedia.com/TERM/s/scalable.html on 18 August 2008.

Wei, Y. Z., Moreau, L. and Jennings, N. R. (2005), A market-based approach to recommender systems, *ACM Transactions on Information Systems (TOIS)* **23**(3), 227–266.

Weinstock, C. B. and Goodenough, J. B. (2006), On Systems Scalability, Technical Note CMU/SEI-2006-TN-012, Software Engineering Institute. Accessed from http://www.sei.cmu.edu/publications/documents/06.reports/06tn012.html on 30 July 2008.

Weyuker, E. and Avritzer, A. (2002), A metric to predict software scalability, *METRICS '02: Proceedings of the 8th International Symposium on Software Metrics* pp.152–158.

Wikipedia (2008), Scalability. Accessed from http://en.wikipedia.org/wiki/Scalable on 18 August 2008.

Williams, L. G. and Smith, C. U. (2004), Web Application Scalability: A model-based approach. Technical report. Software Engineering Research and Performance Engineering Services.

Williams, L. G. and Smith, C. U. (2005), QSEM: Quantitative Scalability Evaluation Method, *in Proceedings of the International Computer Measurement Group*, pp.341–352.

Woodside, M., Petriu, D. C., Petriu, D. B., Shen, H., Israr, T. and Merseguer, J. (2005), Performance by unified model analysis (PUMA), *in WOSP '05: Proceedings of the 5th international workshop on Software and performance*, ACM Press, pp.1–12.

Wu, X. and Woodside, M. (2004), Performance Modeling from Software Components, *in WOSP '04: Proceedings of the 4th international workshop on Software and performance*, ACM Press, New York, USA, pp.290–301.

Xu, D., Riley, G. F., Ammar, M. H. and Fujimoto, R. (2003), Enabling Large-Scale Multicast Simulation by Reducing Memory Requirements, *in PADS'03: Proceedings of the* 17^{th} *workshop on Parallel and distributed simulation*, IEEE Computer Society, Washington, DC, USA, p.69.

Yang, Y., Zhang, J. and Kisiel, B. (2003), A scalability analysis of classifiers in text categorization, *in SIGIR'03: Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, ACM, New York, NY, USA, pp.96–103.

Yen, J. and Tiao, W. A. (1997), A Systematic Tradeoff Analysis for Conflicting Imprecise Requirements, *in RE'97: Proceedings of the 3rd IEEE International Symposium on Requirements Engineering*, IEEE Computer Society, Washington, DC, USA, p.87.

Yokota, H. (2000), Performance and reliability of secondary storage systems, *in WMSCI'00: Proceedings of* 4^{th} *World Multiconference on Systemics, Cybernetics and Informatics*, pp.668–673.

Yu, W. (2008), Scaling Your Java EE Aplications. Accessed from http://www.theserverside.com/tt/articles/article.tss?l=ScalingYourJavaEEApplications on 28th July 2009.

Yuan, M. J. and Sharp, K. (2004), *Developing Scalable Series 40 Applications: A Guide for Java Developers (Nokia Mobile Developer)*, Addison-Wesley Professional.

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

Zhang, R., d. S. Oliveira, B. C., Bivens, A. and McKeever, S. (2007), Scalable problem localization for distributed systems: principles and practices, *in InfoScale'07: Proceedings of the 2nd international conference on Scalable information systems*, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, pp.1–8.

Zhang, X. and Xu, Z. (1995), A semi-empirical approach to scalability study, *SIGMETRICS '95/PERFORMANCE '95: Proceedings of the 1995 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems* **23**(1), 307–308.

Zhang, X., Yan, Y. and Ma, Q. (1994), Measuring and Analyzing Parallel Computing Scalability, *in ICPP'94: Proceedings of the 1994 International Conference on Parallel Processing*, IEEE Computer Society, Washington, DC, USA, pp.295–303.

Zirbas, J. R., Reble, D. J. and van Kooten, R. E. (1989), Measuring the scalability of parallel computer systems, *in Supercomputing'89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, ACM, New York, NY, USA, pp.832–841.