

Visualizing non-functional requirements

Neil A. Ernst, Yijun Yu, and John Mylopoulos
Department of Computer Science
University of Toronto, Toronto, Canada
{nernerst, yijun, jm}@cs.utoronto.ca

Abstract

Information systems can be visualized with many tools. Typically these tools present functional artifacts from various phases of the development life-cycle; these include requirements models, architecture and design diagrams, and implementation code. The syntactic structures of these artifacts are often presented in a textual language using symbols, or a graphical one using nodes and edges. In this paper, we propose a quality-based visualization scheme. Such a scheme is layered on top of these functional artifacts for presenting non-functional aspects of the system. To do this, we use quantified quality attributes. As an example, we visualize the quality attributes of trust and performance among various non-functional requirements of information systems.

Keywords *quality attributes visualization, non-functional requirements, tradeoffs, performance, trust*

1. Introduction

Software visualization supports various phases of the development lifecycle. For example, goal-oriented software *requirements* techniques result in goal structures like those presented in *Objectiver* for the KAOS approach [6] and *OpenOME* for the *i*/Tropos* approach [25]; documented software *designs* are displayed as UML diagrams in various UML editors, such as *Rational Software Architect*; reverse engineered software architectures are shown as boxes and arrows in visualization tools like *Creole* and *lscedit*; the implemented software source code is syntax-highlighted in all modern text editors, such as *VIM*, *Emacs*, and *Eclipse*; and scattered, cross-cutting concerns are plotted as aspects in the *AspectJ* visualizer.

These visualizations focus on the functional artifacts of a software system, ranging from abstract arti-

facts such as requirement goals to very concrete ones that are part of an actual software implementation. As the software artifacts get more detailed, their representations become more complex, making it harder for a viewer to be aware of quality issues – such as maintainability, reusability, security, and trust – without losing focus of primary functional concerns.

Some measures are commonly used for reducing the perceived complexity in visualizations, thereby allowing users to divert attention to wider issues. Many of these are an attempt to implement Shneiderman’s *visual information seeking mantra* – “overview first, zoom and filter, then details-on-demand” [1]. These include scrolling, zooming, the use of color, and various information abstraction mechanisms such as nesting and folding.

These treatments of artifacts do not, however, reveal the tradeoffs among different non-functional aspects of the software systems [17]. They present the *complexity* or size of the system. With regard to the qualities of the software product, such as performance, security, usability, and trust, we need to classify the existing functional artifacts to reflect their quantified representation of *quality attributes*. This requires a metric for each such attribute that can be attached to the functional artifact (whether it is a goal, UML element, or source code). Priorities of such quality concerns can also be highlighted to reveal *bottlenecks*.

In this paper, we generalize the idea used in visualizing a single dimension of concern (performance) to multiple dimensions. We show how visual variables such as color, size, shape, and thickness can be associated with different quantified dimensions. These quality-based visual clues are presented together with the functional artifacts. This helps the human decision maker to assess tradeoffs of quality attributes on the primary functional concern.

The remainder of the paper is organized as follows. Section 2 presents a visual formalism to allow multiple quality concerns to be compared. Section 3 expands this

idea with a case study for visualizing the trust attribute and comparing it with the performance attribute in requirements goal models. Section 4 relates the work with others. Section 5 discusses the results of this approach and concludes by sketching future directions.

2. Visualizing general quality attributes

In visualizing software, one extreme is to show only the functional artifact (e.g. source code): one cannot easily see where to start performance tuning to obtain the most benefit for the minimal effort. Another extreme is to show only the quality attributes in summary: one can not easily see which elements (e.g., variables, loops) of the artifact are the bottleneck for improving the overall quality (e.g. performance). This paper demonstrates a middle-ground approach, integrating quality attributes directly on top of the functional artifacts.

In order to show quality attributes and functional artifacts together, one can operationalize a quality attribute into a part of the functional artifact as an extension to its domain-specific language. Alternatively, one can represent a quality attribute as an orthogonal decoration on top of the functional artifact. We consider the latter on the basis of two properties of the quality attributes.

First, quality attributes are non-functional. Moreover, changes to functional artifacts of a software system are intrusive, *i.e.*, they change the system's functionality. Changes to quality attributes, however, are not. For example, introducing a new language construct such as a new function into a program is intrusive; however, adding a comment to the program does not change its functionality, but may improve its maintainability.

Secondly, a quality attribute tends to be a global concern that crosscuts the entire system. Therefore a language construct for its operationalization, if any, must be either aspect-oriented, or scattered all over the code. Moreover, different quality attributes may interfere with each other, and the designer must make a tradeoff by seeing all of them together. To avoid such clutter, we shall use orthogonal visual clues to encode different quality attributes.

To visualize quality attributes, we use the well-understood notion of visual variables, as introduced by Jacques Bertin [3]. Table 1 shows the mapping between qualities and variables, which include size, shape, texture, orientation, hue, and value (color intensity). Each visual variable has the ability to represent particular information types: for example, size can convey quantitative relationships that orientation cannot. Consequently, an important step in applying visualization is

to correctly identify the attributes of interest. The table presents each quality, followed by the variable we chose, as well as the type of data that quality represents, of which we use two: nominal data, which has no ordering (*e.g.*, country names); and ordinal data, with some predefined total order (*e.g.*, clustered quantities). The last column represents whether the attribute is functional or non-functional. The choice of non-functional artifacts in our case study will be explained in the following section.

3. Visualizing trust in requirements models

In this section, we show the application of one tool to the problem of quality visualization, using OpenOME to visualize a *trust model* in a real-life case study. This motivates our discussion of heuristics for visualizing quality attributes to the implementation of a graph editor for goal-oriented requirements modeling.

3.1. The primary functional artifact

For a goal-oriented requirements engineer, the primary artifact is a goal model [6]. A goal model is a set of *nodes* representing hierarchically decomposed *goals* plus a set of *edges* representing both decompositions and contribution *relationships* among goals. The owners of the goals are called *actors*, which are represented by a *cluster* of goals, with a node on the cluster border indicating the actor *boundary*. The goals that are delegated from one actor to another are represented by edges known as *strategic dependencies* (SD). A SD edge is decorated with an inverted letter "D" to indicate the direction of the dependency. Furthermore, dependencies have types, and the type is represented as a node in the middle of the dependency arc. The part of the graph that represents an actor's internal goals is called the *strategic rationale* (SR). Both SD and SR diagrams present visual models of how particular organizational goals might be met. The system-to-be is defined by the actors that are replaced by a non-human agent, whereas the other actors form the system context. Fig. 1 shows how these are currently displayed in OpenOME.

3.2. Multiple quality attributes

Every goal has quantifiable degrees of fulfillment or denial. Thus two quality attributes *satisficing* (S) and *denial* (D) are associated with each goal. The term *satisficing* is derived from Simon's definition [20], and refers to a property whose satisfaction cannot be fully quantified. Denial is the degree to which a goal cannot be achieved. Both S and D can be normalized as

System property	Visual variable	Data type	Attribute Type
Node type	Shape	Nominal	Functional
Link type	Texture (line)	Nominal	Functional
Node ownership	Position	Nominal	Functional
Performance	Hue	Ordinal	Quality
Certainty	Value	Ordinal (grouped)	Quality
Feasibility	Texture (label)	Quantitative	Quality
Trustability	Size (line)	Quantitative	Quality

Table 1. Visual variables for understanding software qualities in the OpenOME tool.

a real number ranging from 0 to 1. For the leaf-level goals, these attributes represent the feasibility of tasks that comprise detailed requirements for the system-to-be; for the top-level goals, attributes are the rationale answering the initial requirements for the system-to-be. Given these attributes and logic rules, one can apply reasoning algorithms [10, 19] to infer the value of attributes at any given level of the goal model.

Since an actor (component) in the organization may delegate some of the goals to other capable actors, an interesting quality attribute arises, namely, *trust*. Specifically, a trusted delegation has two possible delegatums. One actor can trust or distrust another to execute/fulfill a goal. We term this *DelegateExecution* and it concerns the capability of the delegatee as perceived by the delegator; secondly, one actor can trust or distrust another to access a resource (data). This is termed *DelegatePermission* and concerns the security/privacy of the system [9]. Delegating an execution of a goal implies delegating the trust to access certain resources. When the system is security-critical, the amount of trust must be restrained and the data given to a delegator limited for access.

Distrust is an overhead to the system performance, as every certification involves extra computation as well as human action (e.g. encrypt/decrypt, enter login password, etc.) This presents another quality attribute that can be adjusted for in the analyzed requirements goal model. Therefore, *trust* and *performance* are two quality attributes that influence one another. The trust attribute will control the propagation of the satisficing attributes as a weight-amplifier, whereas the performance attribute will be lower when additional tasks have to be fulfilled due to lower trust.

3.3. Representing a quality attribute as a visual clue

To compare the above-mentioned quality attributes, namely, trust and performance, we can depend on existing algorithms to carry out their computations (such as

[10, 19, 9]).

The task in our application is to visualize the quality attributes on the requirements goal model. It serves two purposes: first, to see what happens as a means of problem determination; secondly, to view the result of the computed attributes when certain requirements of the system change.

We use four quality attributes for our application. Table 1 presents how we mapped these qualities to visual variables in the interface.

The first concerns the degree of *certainty* a particular goal exhibits. Our visualization displays certainty as a brightness of the node background in contrast with the black foreground text (whose Value is 0). Nodes with more legible text have higher values and indicate more certainty. We are still working on choosing sane label placements to preserve readability.

The second dimension concerns the degree of *feasibility* of a particular goal, e.g., is its fulfillment possible? We show this as a text label (e.g. $f = 1.0$) just to the right of the node. This is not typically considered visualization, but given the number of variables we wish to represent, and the limited set of representations, we believe this presents the lowest cognitive overhead. Another possible technique is to resize a node according to the feasibility, but size – area in this case – is not as suitable for quantitative comparison [3].

In our application, the third metric is the *trustability* of a goal when it is delegated from one actor to another. When a goal is decomposed, it has an implicit delegation from the actor to itself, and thus the trust for goals internal to an actor is always 1 (assuming the actor can trust himself to accomplish a task!). To evaluate the overall trust of the actor for the system, one can integrate all the outgoing delegation links that are feasible according to the goal reasoning stage. Trust is displayed as a variation in thickness of a delegation link: thicker lines indicate less trusted (more distrusted) actors.

The fourth metric is the *performance* metric of a goal. As discussed, distrust actually requires additional tasks, either to monitor the delegated execution of a

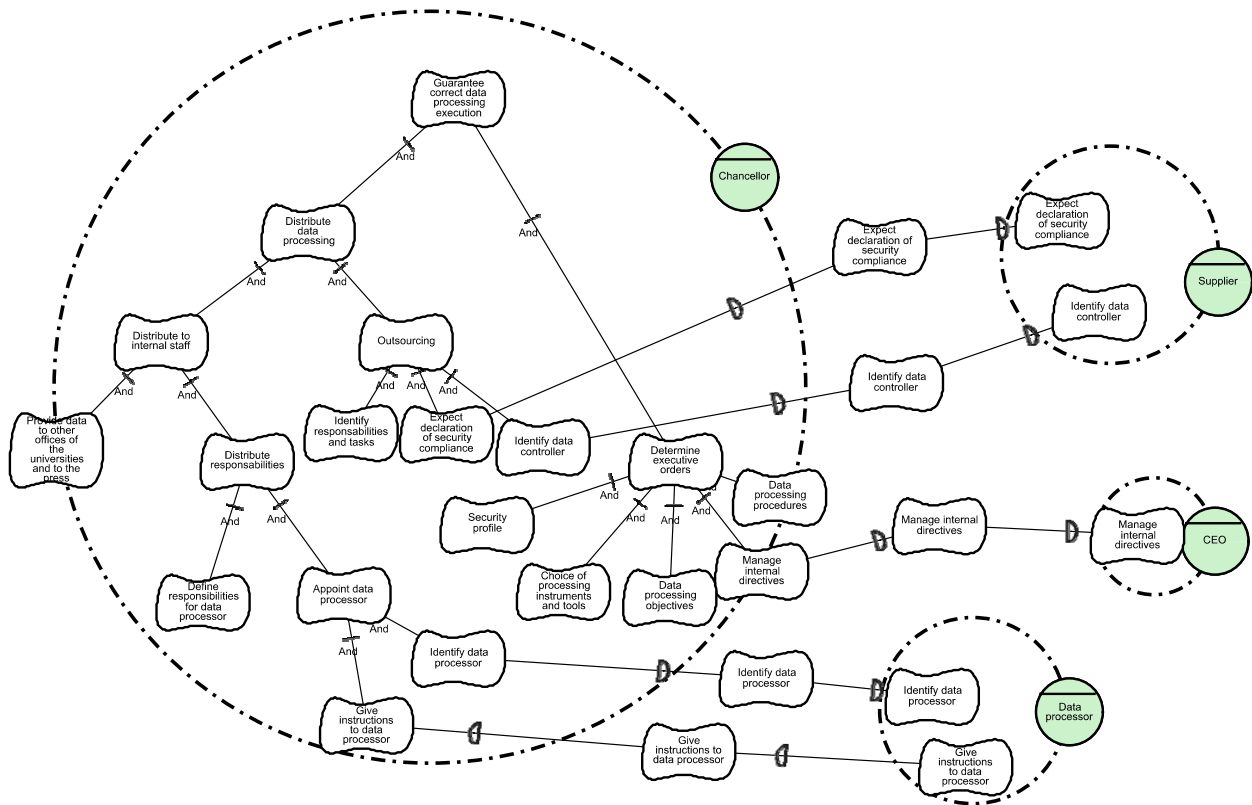


Figure 1. The original dependency model. Curved nodes represent goals; dashed circles are actor boundaries.

goal, or to encrypt/decrypt the resources that are delegated as an access permission. Overall time performance is therefore the sum of existing performance assessment, and the overhead of mitigation tasks. Since the existing performance cost is the same in both cases, we use a ratio of (distrust penalty + normal cost) / normal cost. To display this, a modified version of the Hue variable conveys an idea of “warmth”. That is, the Hue ranges from blue to red, with deeply red nodes representing increased performance overhead.

3.4. Case study

We used a modified version of the OpenOME tool on a trust-based goal model as defined in Giorgini, *et al.* ([9]), to partially validate our proposal. This is a goal model which explains the implications of trust in a dynamic environment. An example of the existing goal model is shown in Fig. 1. The goal model relates actors and goals together, decomposing goals into sub-goals. Each goal can be assigned a desired state, e.g., satisfied, denied, and partial versions of the previous two. The actors in this system are represented as circles encompassing various rationale goals.

3.4.1. Trust and performance. To illustrate our tool, we created a subset of the model for clarity. This subset is shown in Fig. 2. The diagram consists of three actors: a Chancellor and two independent data processors (one of whom we have added to illustrate tradeoff analysis). An actor dependency has an associated goal that must be achieved if there is some distrust level (assuming distrust is modeled as $1 - \text{trust}$). For example, the Chancellor might distrust that Processor #1 will be able to fulfill his goal of “Identify data processor #1”. As a result, he must also “Monitor data processor”, at an additional performance cost. Since Processor #2 is fully trusted, this dependency has no associated performance penalty (yet it may fail other qualities, as we shall show). Links from these goals to the quality goals of trust, performance, certainty, and feasibility are shown using quantified i^* contribution (e.g. helps) notation [24].

While this model is comprehensive, the only way to identify the attributes of a specific node is to bring up a series of links annotated with explicit text labels, which clutter the primary goals and make it difficult to readily understand.

We then add to the original model, adding the trust

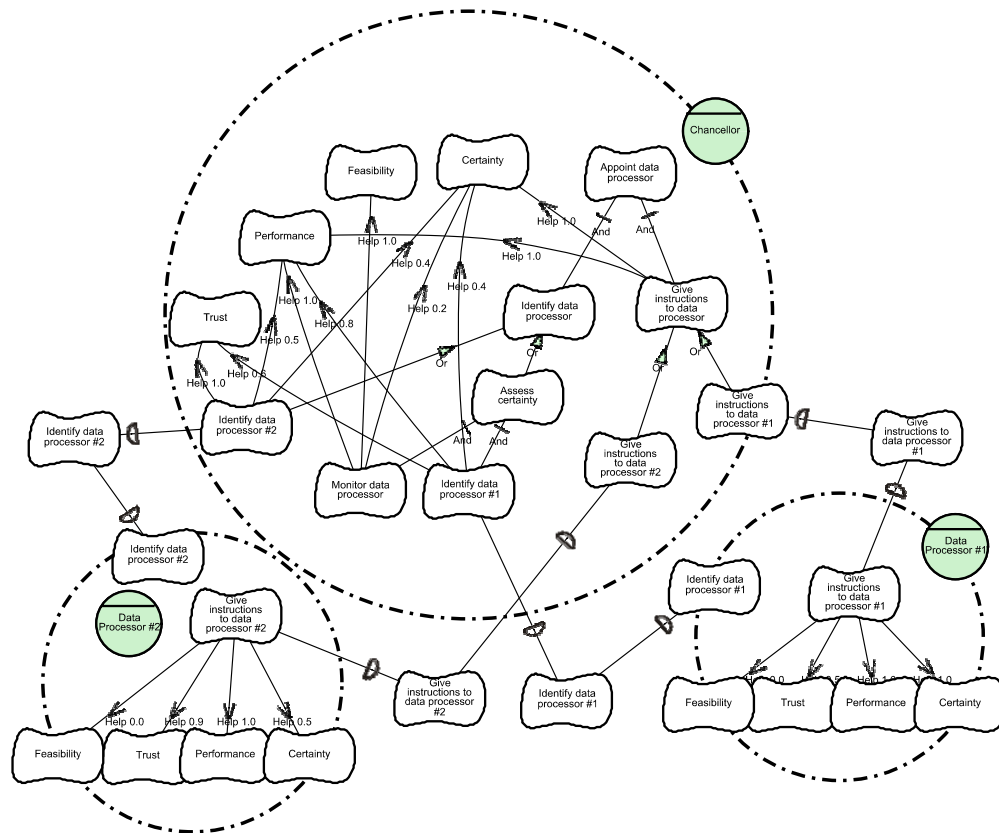


Figure 2. A subset of Fig. 1 showing a new monitoring goal and an alternative processor, as well as additional quality goals.

and performance assessments at the leaf level. For space reasons, we haven't explicitly enumerated the quantities involved. In our scenario, the Chancellor has just started work; Data Processor #1 has a tarnished reputation; Data Processor #2 is more trustworthy, but less feasible (perhaps due to cost, or time to completion).

3.4.2. Representing quality attributes. We manually added the quantitative values in order to represent the quality attributes visually. The rules for doing this are: 1) for certainty and feasibility, we use a min/max "possibilistic" decision procedure [10]; 2) for performance, we take an average of children for "and" branches, and the minimum for "or" branches. Figure 3 shows the result. In future we expect to add a reasoning engine, modeled after that in Giorgini *et al.* [10], which can assess higher-level implications of low-level qualities automatically.

This new representation has several interesting features. Thicker lines indicate less trust, so the thick line from Chancellor to Processor #1 indicates that this is a problematic relationship. We chose the metaphor 'thickness=less' to highlight these potential issues dur-

ing tradeoff analysis. Focusing on the Chancellor's rationale nodes, the "Identify data processor" goal has two alternative subgoals: either to use processor #2, with feasibility 0.5, or to "Assess certainty" of processor #1. From our color variable, this node is dark purple with more legible text, indicating it is more certain, and incurs an extra performance penalty. Processor #2, although less feasible (perhaps he is more costly), is shaded blue, indicating a low or non-existent performance penalty. Although a small example, our quality visualization has nonetheless indicated, quickly and succinctly, a decision tradeoff for the analyst.

4. Theoretical evaluation

This work has not yet been evaluated empirically on large systems. Given its preliminary nature, we instead present a "broad-brush" heuristic walkthrough using Green's cognitive dimensions [12, 13]. We recognize that such an evaluation is not conclusive: see Kutur *et al.* [14] for an example of how results from empirical and theoretical evaluation can differ. Another, similar

Dimension	Description
Abstraction	Types / availability of abstraction mechanisms
Hidden Dependencies	Important links between entities not visible
Secondary Notation	Extra information in means other than formal syntax
Diffuseness	Verbosity of language
Premature Commitment	Constraints on the order of doing things
Viscosity	Resistance to change
Visibility	Ability to view components easily
Closeness of Mapping	Closeness of representations to domain
Consistency	Similar semantics are presented in a similar syntactic style
Error-Proneness	Notation invites mistakes
Hard Mental Operations	High demand on cognitive resources
Progressive Evaluation	Work-to-date can be checked at any time
Provisionality	Degree of Commitment to actions or marks
Role Expressiveness	The purpose of a component is readily inferred

Table 2. Cognitive dimensions used in analysis (from [14])

evaluation mechanism is presented in [5].

Cognitive dimensions “provide a language in which to compare the form and structure (rather than the content) of notations [12]”. The framework was initially targeted at visual programming languages, which feature a high degree of user interaction. However, recent work has also applied it to non-interactive notations, such as UML in [14]. We use the dimensions described in that work for our evaluation. Table 2 defines these.

- **Abstraction** No new abstractions are supported (but see Future Research). Existing abstractions such as the NFR framework are fairly clear to requirements engineers, but new users face a high barrier.
- **Hidden dependencies** Hidden relationships do exist, and need to be shown by model builder. Functional dependencies (such as functional aspects) are hard to see.
- **Secondary notation** Well supported via visual grammar. Position and layout are potentially troublesome.
- **Diffuseness** Scalability remains a concern. We conjecture abstraction will help here.
- **Premature commitment** Goal/softgoal notation enforces non-functional paradigm, which may not be ideal.
- **Viscosity** Low; interface readily supports option exploration and changing the model.

- **Visibility** Ability to scale to larger models is not clear and might affect visibility.
- **Closeness of mapping** A physical mapping of a temperature to a color is a useful analogy to reveal hot spots of the quality attributes.
- **Consistency** Acceptable.
- **Error-proneness** Use of hue/value makes fine distinctions rather difficult. Unfamiliar users may not readily grasp these visual variables.
- **Hard mental operations** For trade-off analysis, the notation supports cognitive operations.
- **Progressive evaluation** As a user’s requirements vary in the priority or preferences for the quality attributes, it is important that he/she be able to separate them, and combine them using different weights. The user can perform an analysis with any amount of information.
- **Provisionality** Exploration is well-supported.
- **Role expressiveness** The principle is to use comparable visual clues for comparing quality attributes because, for example, it is difficult to compare the size of the node with the color of it. Overloading the human ability to understand visual information must be avoided. This implies reducing fine distinctions that make no immediate sense. Rather, these fine distinctions are presented with the empirical data as reinforcement (in the form of labels).

As a result of this evaluation we conclude our provisional notation performs well in the dimensions of viscosity, secondary notations, and supporting hard mental operations. The tool is flexible and accepting of data. However, we do poorly in scalability and abstraction. We cover these aspects in the discussion, below.

5. Related work

There is existing work on software visualization at various levels of the software abstraction hierarchy.

At the execution level, performance visualization tools (such as [28, 26, 27, 4], Intel VTune) show developers where performance bottlenecks occur, such as loop parallelism, frequent cache misses, instruction cycles. Frequently software metrics are combined with program visualization, as in Systa, Yu, and Mueller [23]. At the static, code level, several tools exist, from simple text editors with indented module structures, to complex IDE's such as Eclipse or Visual Studio, to configurable browsing and understanding tools. For example, the Creole tool [22] provides mechanisms for software exploration, concentrating mainly on the source code level – showing class hierarchy, execution flow, package structure, etc. Some tools capture the change as shown through a code repository, tracking software evolution. Moving up the abstraction hierarchy, many tools exist to visualize program architecture. Tools such as Rigi [21] allow users to elide nodes to capture domain abstractions. Work by Langelier *et al.* [15] show non-functional attributes of the system as various visual variables, such as orientation and color. In that sense, they parallel our approach of visualizing non-functional aspects, yet remain concentrated on the source/package level. Perhaps most common are visual UML editors, which either allow for architectural sketching, or may be full round-trip tools (from model to code and back).

At the next level, the requirements level, there are also many tools. Requirements visualization is well-developed, with work that shows complex formal models to domain users [7], or different viewpoints, [8]. Goal modeling tools capture high-level system requirements and constraints, and frequently have graphical modeling tools, such as the OpenOME tool, as shown earlier. Tools in the Tropos project have been designed to capture the process of transitioning from organizational models to architectural design, such as those in Bastos *et al.* [2]. These efforts have not focused on integrating the different visualizations, however.

The i* modeling tool allows people to construct goal models. One project, [11], used Microsoft Excel to present the goal scenarios as tabular options. A tree

was shown visualizing a particular variant; satisfaction scores were mapped from green to red based on the degree of conflict.

Perhaps closest in nature to our proposal are the several tools which provide visualizations for aspect-oriented software development, such as ConcernMapper [18]. These tools attempt to show how various concerns, such as security, logging, etc., are scattered throughout the functional implementation of the code. Our tool, on the other hand, relies on a determination by the developer of which goals (some of which may map to aspects, see [29]) map to which functions.

Finally, there is work on visualizing the multi-dimension quantities mathematically through a Cartesian coordinate systems (3-dimension) or spider diagrams (N-dimension). The problem for such projections into a mathematical domain is that the quantities are not directly attached to the software functional artifacts. Detaching quality attributes from functional artifacts raise an issue of traceability, making it a lot harder for software developers to trace back into their primary concern – the artifacts in a software product.

6. Discussion and Future Research

We identified a lack of quality tradeoff visualizations in software visualization tools and presented several means to present them on top of functional artifacts. The quality visualization framework employs orthogonal visual clues that do not interfere with the primacy of functional artifacts. Visual heuristics were applied to interpret the requirements of an information system in OpenOME. The implemented visualization presents the tradeoffs of quality attributes such as feasibility, certainty, trust, and performance on the requirements goal model of the system.

The real benefit of this general framework for quality attribute visualization is in understanding design tradeoffs in information systems. For example, visualizing bottlenecks is useful for *problem determination* in self-managing autonomous computing systems [16]. One of the objectives of managing adaptive autonomous system is to hide the system complexity without interfering with high-level, strategic decisions. Our tool would allow designers to assess the high-level interactions of non-functional requirements. More research is needed to determine how to evaluate the quantitative implications of such interactions (e.g., evaluation algorithms for goal models; operationalization in specifications).

The generalized approach presented here allows such tradeoffs to be assessed. We use visual variables such as color, size, and position to present the quality attributes of a system directly with the functional artifacts

– in this paper, either source code or requirements models. The only requirement for extending this work is to provide the appropriate metrics for a particular quality. For example, a metricized system usability test could readily be visualized with this tool.

In future, we propose to extend this framework to fully support Shneiderman’s “overview first, zoom and filter, then details-on-demand” visualization process. Finally, as with any tool, we will be conducting user studies to assess viability. These will provide an empirical validation of the existing cognitive dimensions analysis we performed.

From that analysis, we believe our notation fails to address the dimensions of abstraction and scalability (visibility/hidden dimensions). One way to address both issues is to constrain the possible data set. To this end, we are actively researching what we loosely term “goal algebra”. This is a set of object-oriented syntax for the t^* language that will allow for set-theoretic operations prior to displaying the model. Our vision is to integrate these into the tool, and implement quality visualization as part of that framework.

7. Acknowledgements

This work was supported in part by the Natural Sciences and Engineering Research Council.

References

- [1] C. Ahlberg, C. Williamson, and B. Shneiderman, “Dynamic queries for information exploration: an implementation and evaluation,” in *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems*. New York, NY, USA: ACM Press, 1992, pp. 619–626.
- [2] L. R. D. Bastos and J. B. de Castro, “Integration between organizational requirements and architecture,” in *Workshop em Engenharia de Requisitos (WER)*, 2003, pp. 124–139.
- [3] J. Bertin, *Semiology of graphics: diagrams, networks, maps*. Madison, WI: University of Wisconsin Press, 1983.
- [4] K. Beyls, E. H. D’Hollander, and F. Vandeputte, “RD-VIS: A tool that visualizes the causes of low locality and hints program optimizations,” in *International Conference on Computational Science*, vol. 2, 2005, pp. 166–173.
- [5] C. Britton, S. Jones, M. Kutar, M. Loomes, and B. Robinson, “Evaluating the intelligibility of diagrammatic languages used in the specification of software,” in *First International Conference on Theory and Application of Diagrams*, M. Anderson, P. Cheng, and V. Haarslev, Eds. Edinburgh, Scotland, UK: Springer Berlin, September 2000, pp. 376–391.
- [6] A. Dardenne, A. van Lamsweerde, and S. Fickas, “Goal-directed requirements acquisition,” *Science of Computer Programming*, vol. 20, no. 1–2, pp. 3–50, Apr. 1993.
- [7] N. Dulac, T. Viguier, N. Leveson, and M. A. Storey, “On the use of visualization in formal requirements specification,” in *Intl. Conf. on Requirements Engineering*, 2002, pp. 71–80.
- [8] S. Easterbrook and B. Nuseibeh, “Managing inconsistencies in an evolving specification,” in *Second IEEE International Symposium on Requirements Engineering*, 1995, pp. 48–55.
- [9] P. Giorgini, F. Massacci, J. Mylopoulos, and N. Zannone, “Modeling social and individual trust in requirements engineering methodologies,” in *The 3rd International Conference on Trust Management*, Rocquencourt, France, 2005, pp. 161–176.
- [10] P. Giorgini, J. Mylopoulos, E. Nicchiarelli, and R. Sebastiani, “Reasoning with goal models,” in *ER '02: International Conference on Conceptual Modeling*. Tampere, Finland: Springer-Verlag, October 2002, pp. 167–181.
- [11] B. Gonzales-Baixauli, P. J. C. S. Leite, and J. Mylopoulos, “Visual variability analysis for goal models,” in *12th IEEE International Requirements Engineering Conference*, 2004, pp. 198–207.
- [12] T. R. G. Green, “Cognitive dimensions of notations,” in *People and Computers V*, A. Sutcliffe and L. Macaulay, Eds. Cambridge, UK: Cambridge University Press, 1989, pp. 443–460.
- [13] T. R. G. Green and M. Petre, “Usability analysis of visual programming environments: a ‘cognitive dimensions’ framework,” *J. Visual Languages and Computing*, vol. 7, no. 2, pp. 131–174, 1996.
- [14] M. Kutar, C. Britton, and T. Barker, “A comparison of empirical study and cognitive dimensions analysis in the evaluation of uml diagrams,” in *14th Workshop of the Psychology of Programming Interest Group*, J. Kuljis, L. Baldwin, and R. Scoble, Eds., June 2002, pp. 1–14.
- [15] G. Langelier, H. A. Sahraoui, and P. Poulin, “Visualization-based analysis of quality for large-scale software systems,” in *IEEE/ACM International Conference on Automated Software Engineering 2005*, Nov. 2005, pp. 214–223.
- [16] A. Lapouchnian, S. Liaskos, J. Mylopoulos, and Y. Yu, “Towards requirements-driven autonomic system design,” *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, July 2005.
- [17] J. Mylopoulos, L. Chung, and B. Nixon, “Representing and using nonfunctional requirements: A process-oriented approach,” *IEEE Trans. Softw. Eng.*, vol. 18, no. 6, pp. 483–497, June 1992.
- [18] M. P. Robillard and G. C. Murphy, “Concern graphs: finding and describing concerns using structural program dependencies,” in *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)*. New York: ACM Press, May 19–25 2002, pp. 406–416.
- [19] R. Sebastiani, P. Giorgini, and J. Mylopoulos, “Simple and minimum-cost satisfiability for goal models,” in

- Conf. on Automated Information Systems Engineering*, 2004, pp. 20–35.
- [20] H. A. Simon, “Motivational and emotional controls of cognition,” *Psychological Review*, vol. 74, no. 1, pp. 29–39, January 1967.
- [21] M. A. Storey, K. Wong, F. Fracchia, and H. Müller, “On integrating visualization techniques for effective software exploration,” in *Intl Conf. on Information Visualization*, Phoenix, AZ, 1997, pp. 38–45.
- [22] M.-A. D. Storey, D. Cubranic, and D. M. Germán, “On the use of visualization to support awareness of human activities in software development: a survey and a framework,” in *ACM Symposium on Software Visualization*, 2005, pp. 193–202.
- [23] T. Systä, P. Yu, and H. Müller, “Analyzing Java software by combining metrics and program visualization,” in *The Conference on Software Maintenance and Reengineering (CSMR’00)*. Washington, DC, USA: IEEE Computer Society, 2000, p. 199.
- [24] E. S. Yu, “Towards modelling and reasoning support for early-phase requirements engineering,” in *International Symposium on Requirements Engineering*, Annapolis, Maryland, 1997, pp. 226–235.
- [25] E. S. K. Yu, “Modelling strategic relationships for process reengineering,” Ph.D. dissertation, University of Toronto, 1995.
- [26] Y. Yu, K. Beyls, and E. H. D’Hollander, “Visualizing the impact of the cache on program execution,” in *The 5th International Conference on Information Visualization (IV’01)*, 2001, pp. 336–341.
- [27] —, “Performance visualizations using XML representations,” in *The 8th International Conference on Information Visualization (IV’04)*, 2004, pp. 795–800.
- [28] Y. Yu and E. H. D’Hollander, “Loop parallelization using the 3D iteration space visualizer,” *J. Vis. Lang. Comput.*, vol. 12, no. 2, pp. 163–181, 2001.
- [29] Y. Yu, J. C. Leite, and J. Mylopoulos, “From goals to aspects: discovering aspects from requirements goal models,” in *12th IEEE International Requirements Engineering Conference*, Kyoto, Japan, September 2004, pp. 33–42.