

Dealing with system qualities during design and composition of aspects and modules: an agent and goal-oriented approach

Daniel Gross, Eric Yu

Faculty of Information Studies, University of Toronto
{gross, yu}@fis.utoronto.ca

Abstract

Aspects are a recent modularization paradigm that supports the capturing of concerns that usually crosscut the modularization structure of software systems. However, no support is given for representing, capturing, analyzing and tracing how global qualities of the software system, such as performance, maintainability, or extensibility, are addressed during the design and composition of modules and aspects. Intentional agents are novel abstractions for encapsulating design goals and design options of software system artifacts during the design process. This paper proposes the use of intentional agents for capturing and analyzing how global quality concerns are addressed during the design and composition of modules and aspects. The “Intentional aspect” abstraction is introduced for dealing with the design of aspects, the composition of modules and aspects, and for linking aspects to elements within the implementation of the software system. The proposed approach is illustrated through examples taken from an implementation of O-Telos written in MS-Visual Basic and Prolog.

1. Introduction

Most system analysis and design notations and programming languages provide constructs for organizing system descriptions as hierarchical compositions of smaller, modular units, where each unit is designed to take care of one concern or design decision at a time [1, 2]. However, George Kiczales, an originator of the aspect oriented programming approach, observed that these modular units do not clearly capture all the important design decisions that a program must implement. Design decisions are often scattered throughout the modular code, resulting in “tangled” code that is excessively difficult to develop and maintain [2].

Kiczales calls the properties or concerns that these decisions address Aspects. Aspects usually include concerns that involve more than one module. A few examples include concerns such as synchronization, component interaction, persistence, security control, error handling, and structure of data and representation of data. These are usually captured with small code fragments that

are scattered throughout several modules [2]. The goal of aspect-oriented programming is to support programmers in cleanly separating components and aspects from each other, through the provision of mechanisms that make it possible to abstract and compose components and aspects to produce the overall system [2].

Quality requirements, also called Non-functional requirements, are important concerns that are addressed during system design [3-5]. Quality requirements are usually global properties of the software system, which are addressed locally during the design of system modules and aspects. Designers of modules and aspects need to coordinate their design options and choices to ensure that global quality requirements are adequately met. Because the design of aspects potentially affects a plethora of modules, the need for coordinating design choices among aspects and modules is crucial.

Current development environments do not support aspects and modules designers in coordinating and reasoning about their design options and choices to adequately meet the overall quality requirements of the software system. Furthermore, several key traceability issues related to quality requirements need to be addressed when dealing with coordinating the design and implementation of aspects and modules. These include:

- How do design choices within modules or aspects relate to overall system qualities?
- How do aspects and modules composed during implementation affect system qualities?
- What alternative design options were evaluated during the design of aspects and modules, and what where the reason these design options were favored?

This paper proposes a modeling representation based on Intentional agents [6-9] and Intentional aspects to better support coordinating the design of aspects and modules to meet overall quality concerns.

Intentional agents are a means for capturing the intents and the know-how of designers while they design software system artifacts (such as modules and aspects, procedures, data structures, parameter formats, etc.). Intents of designers are captured through design goals, while their know-how is captured through design options that achieve design goals. The term *agent* refers to the *social agency*

involved in the design of software system elements. Intentional agents, in particular, distinguish themselves from software agents, which are descriptions or implementations of computational units of software endowed with some degree of intelligence and autonomy during runtime. Intentional agents are representations of social autonomy designer's have, within a network of other designers, to exercise their know-how and make design choices during design-time [8, 10].

Intentional agents are a descriptive means for capturing design rationales [11]. Models of intentional agents are process-oriented rather than product-oriented [12] because they capture design information for justifying design choices during design, rather than representing design elements that are outcomes of design choices. In fact, intentional agent models do not describe design elements they refer to them within other models. The concept of Intentional aspects, presented within this paper, is based on the concepts of intentional agents.

Intentional agents and aspects are abstractions for encapsulating design goals and design options of modules and aspects, respectively. Design options capture solutions to design goals. Alternative design options are connected through means-ends links to design goals; Softgoals capture quality or non-functional requirements that serve as criteria for choosing among alternatives design options. Dependency links among intentional agents and aspects denote how agents and aspects coordinate quality goals among each other. Composition of aspects with modules is represented as the aggregation of goals and design options found within intentional agent and aspects.

The next section introduces the O-Telos example implementation undertaken by the author. Section three gives examples of aspects within the implementation. Section four illustrates how intentional agent concepts are used to represent these aspects as intentional aspects. Section five illustrates how intentional aspects are linked to implementation code. Section six discusses the proposed approach. Section seven points to related work. Finally section eight concludes and points to future work.

2. O-Telos example implementation

The illustrations for this paper are drawn from modeling and analyzing aspects during the implementation of O-Telos [13]. O-Telos is a frame-based knowledge representation language based on Telos [14]. O-Telos was adapted and enhanced with an underlying proposition representation and a set of axioms to support the implementation of object-oriented deductive databases. Concept-base is an implementation of O-Telos [15] that runs on the Unix based operating system, Solaris. This paper describes another implementation written in

MS-Visual Basic [16] and SWI-Prolog [17], for the purpose of supporting O-Telos on a standalone MS-Windows machine.

The main architecture elements of this implementation can be seen in figure 1. The **Visual Basic** (VB) module includes a Graphical user interface, which is comprised of several **Form** and **GUI event-handling** modules. The form and GUI event-handling modules present user interfaces to the user for defining, retrieving and deleting O-Telos frames, and handle events generated by the user interface, respectively. The VB module includes a **Frame Class** "module" which presents an object interface for creating, retrieving and deleting Frames as well as an **utility** module that offers several routines for submitting frame related commands with parameters through DDE to the SWI Prolog environment.

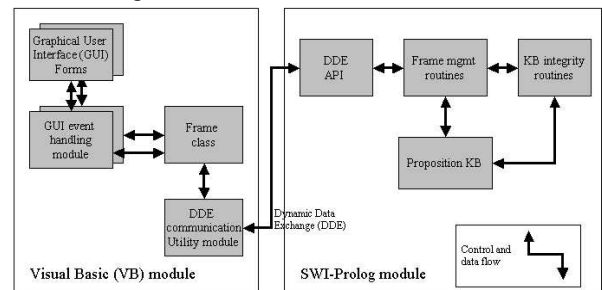


Figure 1: O-Telos/VB High-level architecture

The **SWI Prolog** module includes a **DDE API** module that defines an interface to several Frame and Knowledge base (KB) related commands, and makes them callable via the MS-Windows DDE service. A **Frame management routines** module implements all frame related commands in terms reading, adding and deleting the lower-level propositions representation of frames into the prolog **Proposition KB**. A **KB integrity routine** module checks the integrity of the Proposition KB based on O-Telos 34 principal axioms, when propositions are added or deleted.

3. Aspects within the architecture

Although the architecture in figure 1 suggests a "clean" module structure with clearly separated concerns, there are several unrepresented "cross-cutting" design aspects that span across modules and procedures. For example, how errors are represented and handled, in what format data is passed among modules, how, where and when user entry is checked and validated, and how and when KB data is made persistent. All such concerns have a coding manifestation in several modules. Furthermore, while addressing each of these concerns, several alternative options were considered, tradeoffs in terms of system qualities evaluated and design choices made.

Figure 2 reveals two aspects. The data formats used to pass frame data from the user interface through all

procedures, and the data formats used for result data and error notification. It shows the chain of procedure calls that occurs after the user enters a new frame in the user interface, and how the results “flow” from prolog back to the user interface. The rectangles denote functional procedures or procedure calls, the arrows represent flow of control, or flow of data and the trapezoids represent data items.

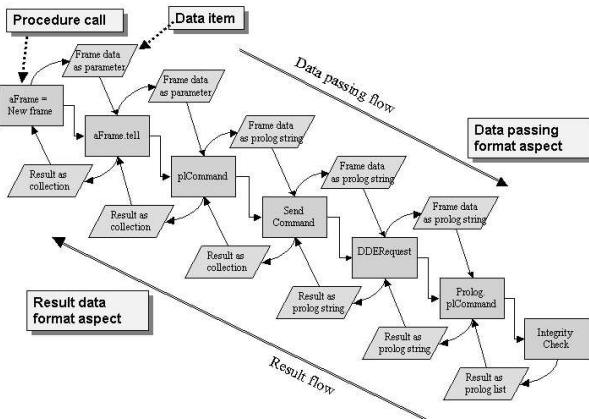


Figure 2: Call chain and data passing for new frame definition

When a user enters a new frame in the user interface, an event routine is triggered within an associated event module (not shown). The event routine creates a new frame object by calling the “aFrame = new frame” statement. It then calls that object’s “tell” method (“aFrame.tell”). The object’s tell method calls the plCommand procedure within the utility module. The plCommand translates the frame data passed so far as parameters into a string representing a prolog list, and calls the sendCommand procedure. The sendCommand procedure submits a “tell” command with the string representation of a prolog list as parameter, via the DDERequest command, to prolog.

Within prolog the DDE API module “receives” the DDERequest command, and calls the plCommand procedure and passes along the prolog list. The plCommand procedure “tentatively” creates a frame, and calls the integrity checker to see whether the newly created frame violates any of the O-Telos axioms. If an axiom is violated the axiom checker returns a prolog list with the prolog atom ‘error’ as the first item, a error code atom as the second item, and error message string as the third item. This prolog list is passed back to the prolog plCommand procedure, which in turn passes it back via the DDE Result variable to the sendCommand procedure in Visual Basic. The sendCommand parses the return string and creates a collection with three items. The first item holds the result code, the second item the error code and the third item the error string. This collection is then passed back to the plCommand, which returns it to the

aFrame.tell procedure, which in turn passes the collection back to the calling event routine. Finally, the event routine checks whether the first item of the collection matches with the string “error”. If this is the case the error string stored as the third collection item is displayed to the user.

Clearly design decisions relating to data passing and data format for returning values and errors crosscut several modules and procedures

4. Representing and composing modules and aspects with intentional agents and aspects

Figure 3 shows an intentional aspect view of the design decisions related to the data-passing format. Circles with a horizontal line across the top third (not shown in figure 3) are called agents and represent existing design artifacts (such as modules or procedures) currently under design. Circles with a horizontal line across the bottom third represent intentional aspects currently under design. Within an intentional agent or intentional aspect, ovals denote functional design goals that the designer need to achieve, cloud like shapes denote softgoals that represent quality or non-functional requirements, which need to be taken into account, while achieving functional design goals. Design options are represented by hexagons, which are related to design goals via means-ends links. Finally design options may contribute positively or negatively to, or make (sufficiently achieve) or break (insufficiently achieve) quality goals. Such contribution links are analyzed when design options are evaluated for how well they meet quality goals.

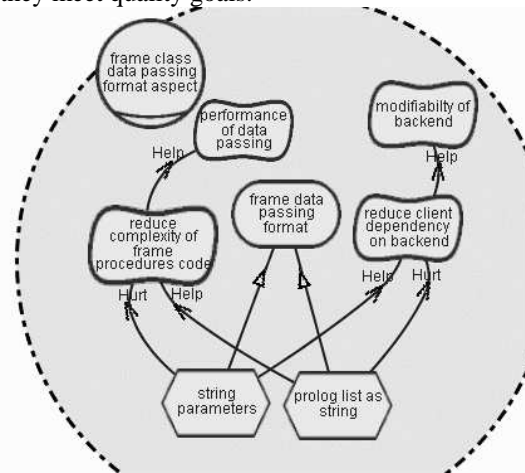


Figure 3: Reasoning about the data passing format within the frame class

In figure 3 the aspect under design is the data-passing format used within the frame class. The **frame class data passing format intentional aspect** includes the **frame data passing format** functional design goal, **reduce client dependency on backend**, and **reduce complexity**

of **frame procedures code** softgoals (quality design goals). Two design options, **prolog list as string**, and **string parameters** are also included. Contribution links between the design options and the softgoals are also shown. The option to use **prolog list as string** as frame data passing format **helps** to **reduce complexity of frame procedures code**, however, it **hurts** the requirement to **reduce client dependency on backend**. The option to use **string parameters** has the opposite effects on the respective softgoals. Note that reducing the complexity of the data passing code helps the **performance of data passing** also, and that reducing the client dependency on the prolog backend helps to achieve **modifiability of the backend**. For example, when instead of using prolog as an implementation language MS-Visual Basic would be used.

Using **prolog list as string** helps to **reduce the complexity of frame procedures code**. This is because the responsibility for creating a prolog string representation for parameters to pass to prolog is delegated to the calling client code, and because the frame class does not need to define additional procedures to deal with the passing of a varying number of parameters. These rationales can be captured through argumentation nodes and links. This modeling feature is not shown in figure3.

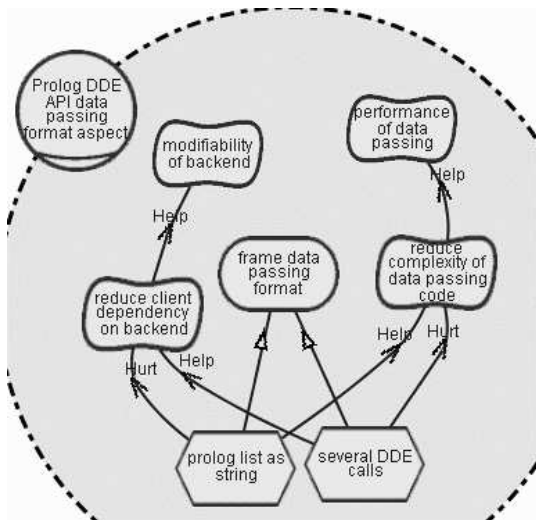


Figure 4: Reasoning about the parameter-passing format within the Prolog DDE API

Figure 4 shows the design considerations related to how frame data is passed through the MS-Windows DDE interface. Two design options were considered – passing a **prolog list as string** to the DDE call, or making **several DDE calls**, one for each parameter that need to be passed to prolog. When **making several DDE calls** helps **reducing the clients' dependency on the prolog backend**. However, it **hurts** the quality goal to **reduce complexity of the data passing code**. The option of using a **prolog list as string** has the opposite effect on the

respective quality goals. Figure 4 also shows that **reducing the complexity of the data passing code** helps the **performance of data passing** through the DDE API, and that **reducing the clients dependency on the backend**, makes the backend modifiable.

Similar diagrams exist for reasoning about the design of the return format used by **sendCommand()** procedure in the utility module and for the **prologAxiomChecker()** procedure within the **KB integrity routines** module. They can be seen in figure 6.

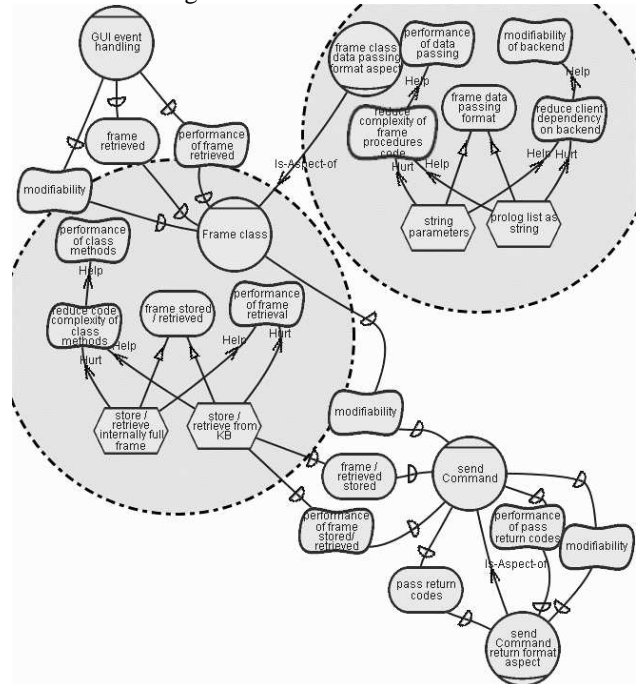


Figure 5: Composing intentional agents and aspects

Figure 5 shows three intentional agents, two intentional aspects and two compositions of intentional agents with intentional aspects. The **Is-Aspect-Of** link between intentional aspects and intentional agents denotes the composition between intentional agents and aspects. Due to space limitations not all intentional agents and aspects discussed in the paper are shown. Some intentional agents and aspects are shown in a collapsed view, hiding their internal design goals and options.

Composing intentional agent and aspects means removing the boundaries between agents and aspects for the purpose of analyzing the aggregation of their quality requirements and their design options. This includes aggregating and jointly evaluating the contribution links directed to *common* quality goals. For example, figure 5 shows that when composing the **frame class** agent with the **frame class data passing** aspect the common **performance** design goals need to be composed and thus

may be affected, depending on the design choices made within **frame class data passing aspect**.

Figure 5 also shows softgoal dependency links between intentional agents and between intentional agents and aspects. Dependency links are directed from a depender to a dependee. Within a dependency the depender specifies a dependuum that becomes a design goal for the dependee [6]. The little symbol on the dependency links is the letter “D”, and shows the direction of the dependency. Through dependency links intentional agents and aspects coordinate the adequate achievement of global quality properties. Dependents give guidance through such softgoal dependency links what local design options dependees should preferred, based on the effect design choices have on the quality goals they specify as dependuum.

5. Linking intentional aspects to the implementation

Modeling intentional agents means reasoning about existing design elements and establishing new ones. This suggests that models of intentional agents are one meta-level removed from models representing design elements and/or from implementation code. Figure 6 shows an intentional agent and aspect diagram for reasoning about the return format, a design element repository and the relationship between intentional agents, the intentional aspects and the implementation artifacts of our application that stored as design elements within the design element repository. Intentional agents have a “refer to” link to design elements. Such a link to a design element means that within the corresponding intentional agent we can find the design goals, quality goals and design options that are related to that design element.

From a methodological point of view when creating such a link between an intentional agent and a design element it means that the design element or some aspect of it are now being designed. It also means that within that intentional agents and possibly related intentional aspects its design goals and design options, which are being considered and evaluated, are made explicit. Design options have either a “generates” or a “refers to” link to design elements. A *generates* link specifies that due to the choices of that design option, a design element has been generated. A *refers to* link specifies that due to the choice of that design option, a design element has been introduced as a consequence of introducing another design element elsewhere during system design.

From a methodological point of view the “generates” link is created from a design option to design elements when a decision is made within its corresponding intentional agent or intentional aspect to adopt that design option. A “refers to” link is created between a design

option and a design element, when the design of that design element refers to that design option, which was previously adopted during the design process within its intentional aspect or module.

Let us describe some of the links depicted in figure 6: The *refers to* link between the **intentional agent prolog axiom checker** and the design element **prolog axiom checker** shows that the prolog axiom checker’s design goals, quality goals and design options are described within its corresponding intentional agent and intentional aspects. The *generates* link between the design option **prolog list** and the design element **return statements** shows that the design of return statements is based on the former design option. The *is-part-of* link between the design element **return statements** and the design element **prolog axiom checker** shows that the return statements are part of the prolog axiom checker procedure. Finally, the *refers to* link between the design option **prolog list** and the design element **parameter indexing statements** shows that the parameter indexing statements are a consequence of the design option adopted previously in the design process.

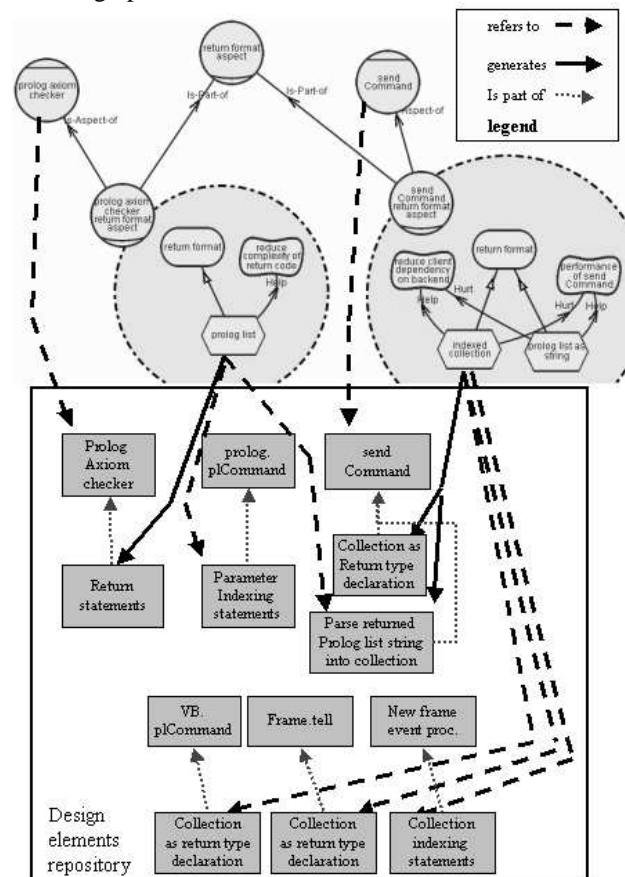


Figure 6: Intentional agents and aspects linked to design elements

Note that the design element **Parse returned prolog list string into collection** has both a *generates* link originating from the design option collection indexed by prolog slots and a *refers* link from the design option prolog list. This means that the implementer of the parsing statements that take a string denoting a prolog list and creates a collection object with indexes corresponding to the prolog list slots, needs to know of return format design decision taken within the prolog axiom checker intentional aspect, whose format was decided within the prolog axiom checker module.

6. Discussion

Traceability and intentional querying support:

During intentional agent and aspect modeling the various links established among modeling elements support several types of traceability:

Through means-ends links established within intentional agents and aspects, designers can trace how and where quality concerns are addressed within aspects, modules or any other design elements of the software system. Furthermore, designers can query where quality concerns are considered and not addressed but traded off with other quality concerns during design. Through dependency links among intentional agents and aspects, designers can trace how global quality concerns are broken down to “lower-level” quality concerns to support and coordinate their achievement within modules, aspects and other design elements.

Through “refers-to” links between intentional agents and design elements, designers can trace back from design elements, to the design rationales of design elements, to the alternative design options considered, and to the rationales for favoring one option over the other. Through “generates” links between design-options within intentional aspects and design elements, designers can navigate to all modules or design elements related to a particular design aspect. They can also trace back from design elements to their design aspect. Through “refers-to” links between design options within intentional aspects and design elements within the design element repository, designers can find all modules and design elements relating to and affected by a design choice related to an aspect. They can also query while establishing new design elements, what design options are related to particular design aspects.

This last querying ability can in particular aid implementers of design elements to quickly retrieve design choices that are relevant to aspects addressed in the software system. For example, an implementer, while writing code to check the return codes of a frame procedure call can be prompted and advised that the return code format is a collection where the first item either equals the string “error” or the string “ok”.

Automated vs. interactive modeling approach:

Aspect-oriented programming aims at creating fully automated “weaving” tools to merge aspects with modules code. For this approach to work, aspect languages are designed, and used by “weaving tools” to merge aspects with modules. AspectJ [18], Cool, and Ridl [19] are examples of such aspect languages designed to provide aspect support for Java, to express synchronization in concurrent OO programs, and to express remote invocation and control the depth of parameter transfer, respectively.

The approach proposed in this paper does not support automated weaving of aspects with modules. Automated weaving is replaced by support for interactively navigating links. Designers or implementer manually link intentional aspects to the design elements in code, and use the intentional models and links to navigate among or query intentions and design elements.

Support for goal-orientation & knowledge-based approach to aspect design:

Intentional aspects are supported by goal structures, which are missing in aspect-oriented programming. Making goals explicit allows for representing, capturing and analyzing alternative design options that address elements within the design of aspects. Having an explicit representation of goals, also allows for a goal-driven approach, where functional and non-functional goals drive the exploration of design options. Current aspect-oriented approaches do not make use of goals as a modeling or programming concept. Having the ability to represent and capture goals explicitly also supports a knowledge-based approach to design. Knowledge of how goals can be achieved can be captured and made available to designers for future reference when similar design goals again need to be addressed.

Cost of the proposed approach: Current design methods, languages and notations focus on representing and modeling the “solution” behaviors or structures that are the outcome of design processes. Additional design rationale and/or requirements traceability notations are offered which usually capture design goals and options “after the fact” – when choices are already made. Capturing such rationale and/or traceability information is then seen as an overhead to the design process. The benefit of such structures is often only experienced at a later stage, during system maintenance and evolution. It is the aim of the modeling approach proposed in this paper to capture such design intents and rationales as an integral part of the design deliberation and decision-making processes during module and aspect design. The designers who use the notation should experience value in dealing with quality requirements during the design of modules and aspects, and during the coordination efforts when dealing with system wide quality requirements. This should reduce the overhead experienced when establishing and maintaining these structures over time.

7. Related work

Ramesh and Jarke [20] consolidate requirements traceability needs and practices of 26 development organizations, into two reference models – a simple low-end and a complex high-end traceability model.

Several distinctions between the approach proposed in this paper and the proposed requirements traceability Ramesh and Jarke's reference models can be drawn:

Distinguishing between functional and non-functional requirements: The proposed approach in this paper is based on the NFR Framework [21, 22], which distinguishes between functional and non-functional requirements (NFRs). The NFR framework treats NFRs as potentially conflicting or synergistic goals to be achieved, and are used to guide and rationalize the various design decisions during system/software development. Furthermore, because NFRs, such as maintainability, usability and security, are often subjective by nature, they are often achieved not in an absolute sense, but to a sufficient or satisfactory extent (the notion of satisficing [23]). Accordingly, the NFR Framework introduces the concept of softgoals, whose achievement is judged by the sufficiency of contributions from other (sub) softgoals. Throughout the development process, consideration of design alternatives, analysis of design tradeoffs and rationalization of design decisions are all carried out in relation to the stated softgoals and their refinements.

The concepts of softgoals, and the contribution types among softgoals provide a “softer” notion for specifying and reasoning about NFR achievement, than is provided by the reference models proposed by Ramesh and Jarke. They do not make any distinctions between functional and non-functional requirements, when linking Requirements to design objects.

Dealing with Aspects: The reference models proposed by Ramesh and Jarke do not provide support for representing aspects as an independent ontological entity for linking requirements to aspects and aspects to modules. This approach recognizes the need to deal with aspects as “first class citizens” to support the organizing of crosscutting design and code, and to link such design and code to functional and non-functional requirements.

Encapsulating design goals vs. open design space: A key feature of intentional agents and aspects is the ability to encapsulate design goals within agents or aspect boundaries, and to specify goal dependencies among them. Goal dependencies enable specifying what quality design goals designers need to consider and what goals they can safely ignore during the design of modules and aspects. Intentional agent or aspect boundaries “shield” designers of modules and aspects from system design goals not relevant to the design task at hand. This can be seen as loosening Parnas information hiding principles [1,

24]. It supports specifying how design choices made within modules and aspects have an effect on other parts of the system but without exposing the detailed knowledge of the design options and choices within the boundaries of intentional agents and aspects.

The reference models proposed by Ramesh and Jarke view all traceability links as residing within a global design space, where links can be drawn from any design item to any requirement item. Thus no support for partitioning and hiding parts of the design space in terms of NFRs and design option explorations is given.

This work draws from Eric Yu's work on intentional agents. Yu first defines the notion of intentional agent in his thesis on process reengineering [6]. In his thesis intentional agents are the focal point for modeling intents and rationales behind process design within organizations. Yu, Gross and Chung further develop the idea of intentional agents for the use of software design modeling [7-9, 25].

The Tropos project [26] makes use of the intentional agents concept for specifying and reasoning about operational goals during requirement specification. In Tropos intentional agents have a behavioral and operational meaning rather than a strategic and descriptive meaning described in this paper. Aspects are not addressed in Tropos.

The design repository was adopted from the DAIDA project [27], which offers a comprehensive repository for capturing and navigating among analysis, design and implementation knowledge.

8. Conclusion and future work

This paper presents intentional aspects, a novel concept for dealing with system qualities during the design and composition of aspects and modules. Intentional aspects are also used for navigating to and among aspects within the implementation and for navigating from aspects within the implementation to related quality design goals. The intentional aspect concept is illustrated through an example implementation of O-Telos in MS-Visual Basic and SWI-Prolog.

Open research questions exist on issues related to non-functional requirements (NFRs). Should NFRs appear within intentional aspects only, or are NFRs “full fledged” intentional aspects themselves.

Other important issue to address are: How quality concerns are decomposed within intentional agents and aspects to “lower-level” quality concerns that are then delegated through dependency links to other intentional agents and aspects. How to compose design goals that qualify different design elements in the design element repository when integrating intentional agents and aspects, needs further analysis.

Future work can focus on investigating how intentional aspect modeling environments can be seamlessly integrated into current software design and development environments. Future work can also focus on how to keep quality goal information attached to modules and aspects during the execution of the software system. Quality design goals can be used to reason about composing modules and aspects during runtime such as when dynamically composing web-services and mobile agent code within distributed computational environments.

Further case studies need to be performed to gain more experience with intentional agent and aspect, to find their limitations and, where needed, propose additional traceability constructs. Future work can also focus on developing (semi) automated support for code generation and for linking intentional aspects to design elements.

Acknowledgements

Financial support from Communications and Information Technology Ontario (CITO), the Natural Sciences and Engineering Research Council of Canada (NSERC), and Mitel Corporation are gratefully acknowledged. We also like to thank the anonymous reviewers for their valuable comments.

References

- [1] D. L. Parnas, "Information Distribution Aspects of Design Methodology," Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania February 1971.
- [2] K. Czarnecki and U. Eisenecker, *Generative programming : methods, tools, and applications*. Boston: Addison Wesley, 2000.
- [3] S. Keller, E. L. Kahn, G., and R. Panara, B., "Specifying Software Quality Requirements with Metrics," in *Tutorial: System and Software Requirement Engineering*, R. Thayer, H. and M. Dorfman, Eds.: IEEE Computer Society Press, 1990, pp. 145-163.
- [4] D. Gross and E. Yu, "From Non-Functional Requirements to Design through Patterns," *Proceedings of the 6th International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ'2000)*, 2000.
- [5] B. W. Boehm, *Characteristics of software quality*. Amsterdam, New York: North-Holland Pub. Co. :American Elsevier, 1978.
- [6] E. Yu, "Modeling Strategic Relationships for Process Re-Engineering," in *Department of Computer Science*. Toronto: University of Toronto, 1994a.
- [7] E. Yu, "Agent Orientation as a Modelling Paradigm," *Wirtschaftsinformatik*, vol. 43, pp. 123-132, 2001a.
- [8] E. Yu, "Agent-Oriented Modelling: Software Versus the World," *Agent-Oriented Software Engineering AOSE-2001 Workshop Proceedings*, 2001b.
- [9] D. Gross and E. Yu, "Evolving System Architecture to Meet Changing Business Goals: an Agent and Goal-Oriented Approach," *Proceedings of the First International Workshop From Software Requirements to Architectures (STRAW 2001)*, 2001b.
- [10] M. Woodlridge and P. Ciancarini, "Agent-oriented software engineering: The state of the art," in *Handbook of software engineering and knowledge engineering*: World Scientific Publishing, 2001.
- [11] J. Lee, "Design Rationale Systems: Understanding the Issues," *IEEE Expert*, pp. 78-85, 1997.
- [12] J. Mylopoulos, L. Chung, and B. Nixon, "Representing and using nonfunctional requirements: a process-oriented approach," *IEEE Transactions on software engineering*, vol. 18, 1992.
- [13] M. Jeusfeld, "Aenderungskontrolle in deduktiven Objektbanken," in *Fakultaet fuer Mathematik und Informatik*. Passau zur Erlangen: Universitaet Passau zur Erlangen, 1992.
- [14] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis, "Telos: Representing Knowledge About Information Systems," *ACM Transactions on Information Systems*, vol. 8, pp. 325-362, 1990.
- [15] Concept-Base, "<http://www-i5.informatik.rwth-aachen.de/CBdoc/cbflyer.html>." 2002.
- [16] Visual-Basic, "<http://msdn.microsoft.com/vbasic/>." 2002.
- [17] SWI-Prolog, "<http://www.swi-prolog.org/>." 2002.
- [18] aspectj.org, "<http://www.aspectj.org/>." 2002.
- [19] C. V. Lopes, "A language framework for distributed programming," in *Graduate School of College of Computer Science*. Boston, MA: Northwestern University, 1997.
- [20] B. Ramesh and M. Jarke, "Towards Reference Models for Requirement Traceability," *IEEE Transactions on software engineering*, vol. 27, 2001.
- [21] L. Chung, "Representing and Using Non-Functional Requirements for Information System Development: A Process-Oriented Approach," in *Department of Computer Science*. Toronto: University of Toronto, 1993.
- [22] L. Chung, B. Nixon, E. Yu, and J. Mylopoulos, *Non-functional requirements in software engineering*. Boston: Kluwer Academic, 2000.
- [23] H. A. Simon, *The Science of the Artificial*. Cambridge, MA: The MIT Press, 1981.
- [24] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, pp. 1053-1058, 1972.
- [25] L. Chung, D. Gross, and E. Yu, "Architectural design to met stakeholder requirements," in *Software Architecture*, P. Donohue, Ed. San Antonio, Texas, USA: Kluwer, 1999, pp. 545-564.
- [26] J. Castro, M. Kolp, and J. Mylopoulos, "A Requirement-Driven Software Development Methodology," *Proc of the 13th International Conference on Advanced Information Systems Engineering CAiSE 01*, 2001.
- [27] M. Jarke, J. Mylopoulos, W. Schmidt, and Y. Vassiliou, "DAIDA: An Environment for Evolving Information Systems," *ACM Transactions on Information Systems*, vol. 10, pp. 1-50, 1992.