ACSE 2004 PROCEEDINGS

4th International Workshop on Adoption-Centric Software Engineering

25 May 2004 • Edinburgh, Scotland, UK

Workshop at
26th IEEE/ACM International Conference on
Software Engineering (ICSE 2004)

ACSE 2004 PROCEEDINGS

Table of Contents

Evaluation and Applications

Interoperability, Integration, and Synchronization

Organizing Committee

Dr. Robert Balzer, Teknowledge Corporation, USA
After several years at the Rand Corporation, Dr. Balzer left to help form the University of Southern California's Information Sciences Institute (USC-ISI) where he served as Director of ISI's Software Sciences Division and Professor of Computer Science at USC. In 2000 he joined Teknowledge Corporation as their CTO and Director of their Distributed Systems Unit, which combines AI, DB, and SE techniques to automate the software development process. His current research includes wrapping COTS products to provide safe and secure execution environments, extend their functionality, and integrate them together; instrumenting software architectures; and generating systems from domain specific specifications.

Dr. Marin Litoiu, IBM Canada Ltd., Canada
Dr. Litoiu is member of the Centre for Advanced Studies at the IBM Toronto Laboratory where he initiates and manages joint research projects between IBM and Universities across the globe in the area of Application Development Tools. Prior to joining IBM (1997), he was a faculty member with the Department of Computers and Control Systems at the University Politechnica of Bucharest and held research visiting positions with Polytechnic of Turin, Italy, (1994 and 1995) and Polytechnic University of Catalunia (Spain), and the European Center for Parallelism (1995). Dr. Litoiu's other research interests include distributed objects; high performance software design; performance modeling, performance evaluation and capacity planning for distributed and real time systems.

Dr. Hausi A. Müller, University of Victoria, Canada
Dr. Müller is a Professor at the University of Victoria, Canada. He is a Visiting Scientist with the Centre for Advanced Studies at the IBM Toronto Laboratory and the Carnegie Mellon Software Engineering Institute. He is a principal investigator of CSER and Chair of its Technical Steering Committee. Together with his research group he investigates methods and technologies to build adoption-centric software engineering tools and to migrate legacy software to object-oriented and network-centric platforms. Dr. Müller's research interests include software engineering, software evolution, reverse engineering, software reengineering, program understanding, software engineering tool evaluation, software architecture, and the generation of software engineering tools using Scalable Vector Graphics (SVG) technology. He was General Chair for ICSE 2001 & IWPC-2003 and Program Co-Chair for CASCON 2003.

Dr. Dennis B. Smith, Carnegie Mellon Software Engineering Institute, USA

Dr. Dennis B. Smith is the Lead for the SEI Initiative on the Integration of Software Intensive Systems. This initiative focuses on addressing issues of interoperability and integration in large scale systems and systems of systems. Earlier, he was the technical lead in the effort for migrating legacy systems to product lines. In this role he developed the method Options Analysis for Reengineering (OAR) to support reuse decision-making. Dr. Smith has also been the project leader for the CASE environments project. This project examined the underlying issues of CASE integration, process support for environments and the adoption of technology. Dr. Smith has been the lead in a variety of engagements with external clients. He led a widely publicized audit of the FAA's troubled ISSS system. This report produced a set of recommendations for change, resulting in major changes to the development process, and the development of an eventual successful follow-on system. Dr. Smith is a co-author of the book, Principles of CASE Tool Integration, Oxford University Press, 1994. He has published a wide variety of articles and technical reports, and has given talks and keynotes at a number of conferences and workshops. He is also a co-editor of the IEEE recommended practice on CASE Adoption. He has been general chair of two international conferences, IWPC '99 and STEP '99.

Dr. Margaret-Anne Storey, University of Victoria, Canada

Dr. Storey is an Associate Professor at the University of Victoria. Her main research interests involve understanding how people solve complex tasks, and designing technologies to facilitate navigating and understanding large information spaces. With her students and she is working on a variety of projects within the areas of software engineering, human-computer interaction, information visualization, social informatics and knowledge management. Dr. Storey is a fellow of the ASI and as such collaborates with the IBM PDC on HCI issues for eCommerce and distributed learning applications, and with ACD systems. She is a principal investigator for CSER developing and evaluating software migration technology and a visiting researcher at the IBM Centre for Advanced Studies.

Dr. Scott R. Tilley, Florida Institute of Technology, USA

Dr. Tilley is an Associate Professor in the Department of Computer Sciences at the Florida Institute of Technology, and Principal of S.R. Tilley & Associates, an information technology consultancy based on Florida's Space Coast. He has a Ph.D. from the University of Victoria. He is Chair of the Steering Committee for the IEEE Web Site Evolution (WSE) series of events, and the current President of the Association for Computing Machinery's Special Interest Group on the Design of Communication (ACM SIGDOC).

Dr. Kenny Wong, University of Alberta, Canada

Dr. Wong is an Assistant Professor at the University of Alberta. His main areas of research are software architecture, integration, evolution, and visualization. This research includes conducting case studies, building and using integrated environments for reverse engineering, and exploring collaborative program understanding of heterogeneous systems. Current industrial collaborations include IBM and klocwork Inc. He is a principal investigator of CSER and ASERC. He co-manages a Canadian Foundation for Innovation facility to study collaborative software development and issues of system diversity. Dr. Wong was also Program Chair for IWPC 2003 and WSE 2003.

Sponsors and Supporters

Sponsoring Organizations

Supporting Organizations

Carnegie Mellon Software Engineering Institute, USA
Florida Institute of Technology, USA
IBM Center for Advanced Studies, Canada
Teknowledge Corporation, Inc., USA
University of Alberta, Canada
University of Victoria, Canada

ACSE 2004: May 25, 2004; Edinburgh, Scotland, UK

www.acse2004.cs.uvic.ca

4th International Workshop on Adoption-Centric Software Engineering

Bob Balzer
Teknowledge Corp.
balzer@teknowledge.com

Marin Litoiu
IBM Canada Ltd.
marin@ca.ibm.com

Hausi Müller
University of Victoria
hausi@cs.uvic.ca

Dennis Smith
Carnegie Mellon
dbs@sei.cmu.edu

Margaret-Anne Storey
University of Victoria
mstorey@cs.uvic.ca

Scott Tilley
Florida Institute of Technology
stilley@cs.fit.edu

Kenny Wong
University of Alberta
kenw@cs.ualberta.ca

Abstract

The ACSE series of events aims to advance the adoption of software engineering tools and techniques by bringing together researchers and practitioners who investigate novel approaches to fostering the transition between limited-use research prototypes and broadly applicable practical solutions. One proven technique to aid adoption is to leverage existing commercial platforms and infrastructure. The key objective of ACSE 2004 is to explore innovative approaches to the adoption of proof-of-concept systems by embedding them in extensions of Commercial Off-The-Shelf (COTS) products and/or using middleware technologies to integrate the prototypes into existing toolsets.

Keywords: adoption-centric, COTS, middleware, software engineering, tools

1. Introduction

Research tools in software engineering often fail to be adopted and deployed in industry. Important barriers to adopting these tools include their unfamiliarity with users, their lack of interface maturity, their limited support for complex work products of software development, their poor interoperability, and their limited support for the realities of system documentation engineering. Developing and deploying innovative research tools and ideas as extensions to modern, commonly used platforms may ease these barriers.

ACSE 2004 is the fourth workshop in the series of events focused on Adoption-Centric Software Engineering (ACSE). The series started with an IBM CASCON workshop on Adoption-Centric Tool Development (ACTD) in Toronto in 2001 [1]. This first workshop grew out of a CSER (Consortium of Software Engineering Research) research project on Adoption-Centric-Reverse Engineering (ACRE) [2]. The workshop quickly evolved into the Adoption-Centric Software Engineering Workshop (ACSE 2002), which was held at STEP 2002 in Montréal [3]. ACSE 2002 was followed by the third ACSE workshop, ACSE 2003, which was held at ICSE 2003 in Portland [4].

In addition to ACSE, there are several other initiatives that discuss issues of technology adoption. For example, the SEI COTS-Based Systems (CBS) Initiative [6] and workshops it sponsors, and the recent 1st International Workshop on Incorporating COTS-Software into Software Systems: Tools and Techniques (IWICSS) [7].

2. ACSE 2003

ACSE 2003 was a great success, with over 30 participants and a 116-page Proceedings of position papers that was published as an SEI Special Report [5]. The workshop comprised the following five sessions:

- Problems: Adoption challenges, issues, and factors
- Theories: Adoption models and cognitive support
- Applications: Effective development, authoring, and learning environments
- Techniques: Tool, interoperability, integration, and extension
- Lessons learned: Case studies and experiences

Adoption is typically an important factor in software engineering research, but usually not the most important

requirement. In fact, investigators might argue that thinking too much about adoption hampers innovation in research. The projects discussed in the ACSE 2003 proceedings illustrate that by making adoption a key or even the most important requirement can lead to truly innovative and surprising research results and solutions.

Elevating adoption to the top of the requirements heap or quality criteria heap and, as a result, affecting innovation is one of the central ideas of ACSE. Making adoption a key requirement and focusing on non-functional requirements [8] are not new ideas. However, concentrating one key issue affords lateral thinking that may lead us away from traditional solutions and towards radically different solutions. Note that the same approach of isolating a key requirement or quality criterion to facilitate innovative solutions might work just as well for such diverse non-functional requirements as security, autonomic computing, cognitive support, generation, extensibility, or interoperability [9].

3. ACSE 2004: Theme & Objectives

Every year, the software engineering research community produces numerous prototypical tools, but few enjoy the benefits of having the practitioner community investigate, evaluate, and validate their efficacy. The ACSE series of events aims to advance the adoption of software engineering tools and techniques by bringing together researchers and practitioners who investigate novel approaches to fostering the transition between limited-use research prototypes and broadly applicable practical solutions.

One proven technique to aid adoption is to leverage existing commercial platforms and infrastructure. Users will more likely adopt tools that work in an environment that they use daily and know intimately. For example, common office suites are used to browse Web content, produce multimedia documents, prepare presentations, and maintain budgets.

The key objective of ACSE 2004 is to explore innovative approaches to the adoption of proof-of-concept systems by embedding them in extensions of Commercial Off-The-Shelf (COTS) products and/or using middleware technologies to integrate the prototypes into existing toolsets.

In the long run, improved understanding of adoption issues by academic and industrial software engineering tools developers will ease the transition of software engineering research results to industrial practice.

References

[1] H.A. Müller, M. Litoiu, A. Weber, and K. Wong, "Workshop on Adoption-Centric Tool Development (ACTD)," IBM CASCON, Toronto, Canada, Oct 2001. https://www-927.ibm.com/ibm/cas/archives/2001/workshops/descriptions37.shtml

[2] H.A. Müller, M.-A. Storey, and K. Wong; "Leveraging Cognitive Support and Modern Platforms for Adoption-Centric Reverse Engineering (ACRE)," CSER Research Project, Nov. 2001. http://www.acse.cs.uvic.ca

[3] Tilley, S.; Müller, H.; O'Brien, L.; and Wong, K. "Report from the Second International Workshop on Adoption-Centric Software Engineering (ACSE 2002)". *Proceedings of the 10th International Conference on Software Technology and Engineering Practice* (STEP 2002: October 6-8, 2002; Montréal, Canada), pp. 74-81. Los Alamitos, CA: IEEE Computer Society Press, 2003.

[4] R. Balzer, J.-H. Jahnke, M. Litoiu, H.A. Müller, D.B. Smith, M.-A. Storey, S.R. Tilley, and K.Wong; "3rd International Workshop on Adoption-Centric Software Engineering (ACSE 2003)," pp. 789-790. *Proceedings 25th IEEE/ACM International Conference on Software Engineering* (ICSE 2003: Portland, OR; May 3-10, 2003).

[5] R. Balzer, J.-H. Jahnke, M. Litoiu, H.A. Müller, D.B. Smith, M.-A. Storey, S.R. Tilley, K.Wong, and A. Weber, (eds.); *Proceedings of the 3rd International Workshop on Adoption-Centric Software Engineering* (ACSE 2003: May 9, 2003; Portland, OR). Published as CMU/SEI-2003-SR-004. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, June 2003. ftp://ftp.sei.cmu.edu/pub/documents/03.reports/pdf/03sr004.pdf

[6] SEI CBS Group. "COTS-Based Systems (CBS) Initiative," CMU SEI, 2004. http://www.sei.cmu.edu/cbs/index.html

[7] A. Egyed and D.E. Perry, (eds.); Proceedings 1st International Workshop on Incorporating COTS-Software into Software Systems: Tools and Techniques (IWICSS), Retondo Beach, California, February 2004. Online at http://www.tuisr.utulsa.edu/iwicss/IWICSS-2004_Proceedings.pdf

[8] L. Chung, B. Nixon, E. Yu, and J. Mylopoulos. "Non-functional Requirements in Software Engineering," Kluwer Academic Publishing, 1999.

[9] H.A. Müller. "Lateral Mining in Legacy Systems: Quality-Specific Asset Mining," Presentation at CASCON 2003 Workshop on Mining Legacy Assets organized by D.B. Smith, K. Kontogiannis, L. O'Brian, and W. Huang. https://www-927.ibm.com/ibm/cas/archives/2003/workshops/mining.shtml

Adoption Centric Software Evolution

Ying Zou[1] and Kostas Kontogiannis[2]

Dept. of Electrical & Computer Engineering
Queen's University[1]
Kingston, ON, K7L 3N6, Canada
Ying.Zou@ece.queensu.ca

University of Waterloo[2]
Waterloo, ON, N2L 3G1, Canada
kostas@swen.uwaterloo.ca

Abstract

Software systems are continuously being evolved to meet their user's changing requirements. To prevent the decline of software quality during the evolution, it is critical to aim for an adoption-centric evolution process. Adoption-centric evolution refers to evolution activities that allow for target requirements and adoption criteria to be considered as part of the evolution and system transformation process. In this position paper, we aim to address three major issues for such an adoption-centric software evolution framework, namely, software quality modeling, its operationalization in the evolution process, and its evaluation in the new migrant system.

1. Introduction

A software system usually undergoes many and complex maintenance activities throughout its life-cycle, such as correcting faults, improving performance, adapting the product to a new environment, or adding new functionality. As Lehman pointed out in his laws of software evolution, "software which is used in real-world environments must change or become less and less useful in that environment [1]. Furthermore, "as an evolving program changes, its structure becomes more complex, unless active efforts are made to avoid this phenomenon" [1]. Software re-engineering is one of the research areas that aim to manage software evolution. Generally, it involves the analysis and transformation of a system to reconstitute it in a new form [2].

For the new system to be adopted in its new environment, specific quality and functional characteristics should be met. In most cases, re-engineering and evolution activities are planned with specific target objectives in mind. In this context, adoption-centric software evolution refers to a re-engineering process that aims to produce new migrant systems that satisfy specific adoption criteria such as enhanced maintainability, portability, security, performance, or reliability.

To this extend there are always questions raised, concerning whether the newly resultant system would or could possess as good characteristics as the original system or even better ones (e.g., whether the new system will perform as fast as the old system or whether the new system will be as portable and scalable as the old one). To address such questions, software evolution activities should not occur in a vacuum and it is important to adopt quality requirements in the maintenance process.

Generally, quality requirements, such as reusability, maintainability, performance, portability and, security define system properties, constraints and software qualities of the system being developed or maintained. Maintaining qualities of an evolving software system at satisfactory levels is a challenging task, especially, when the functionality is deeply embedded in the existing source code and spread out in various physical locations. To prevent the qualities of evolving software from deteriorating, changes to it should be understood. In this respect, the system may conform to specific target objectives such as better performance and higher maintainability.

In this position paper, we aim to address three major issues to achieve adoption-centric software evolution. These include: 1) denoting software quality models, 2) incorporation of quality models into the evolution process, and 3) validation of quality characteristics and adoption criteria in the migrant system. The desired quality goals and adoption criteria can be elicited based on domain knowledge, customer interviews and, system documentation. In this context, it is critical to effectively model high level quality requirements in a way that facilitates both the reengineering process and the validation of whether the quality goals have been achieved in the migrant system. For migrating a system, a set of transformations can be applied to alter code features and improve the qualities of the new system. For example, source code features such as pervasive use of *gotos* can be problem prone and may violate target objectives and adoption criteria. Each transformation can be selected and applied according to its potential impact on the desired qualities in the new resultant system. To assess whether the desired goals have been achieved in the newly revolved system, a set of software metrics and source code features are evaluated.

The rest of the paper is organized as follows. Section 2 describes software quality modeling process. Section 3 discusses the techniques to manage quality requirements over the evolution process. Section 4 presents the quality evaluation methods. Finally, Section 5 concludes the paper.

2. Software Quality Modeling

To guide the evolution process to meet quality objectives, we provide a software quality modeling process that elicits and denotes quality goals in a quantitative way. The proposed quality and adoption centric evolution process can be presented by the following steps.

1) *Assess quality status:* a software quality assessment is an attempt to analyze and describe a software system's quality from different perspectives: its characteristics, strengths and weaknesses [3]. Especially, for the evolution process, we aim to preserve the system's behavior while enhancing its quality. Software metrics are widely adopted for quality assessments. Moreover, the result of the assessments facilitates the prediction of evolution efforts.

2) *Identify critical quality bottlenecks:* the critical quality bottlenecks refer to error prone areas or characteristics that prevent the system of being efficiently maintained or extended. The critical bottlenecks are subjective, and specific to the target domain. In particular, different bottlenecks may lead to conflicts towards achieving desired qualities goals. In this context, the key bottlenecks can be rated based on the domain knowledge.

3) *Establish quality objectives:* quality objectives are generally sufficient for addressing serious quality problems, such as bottlenecks, at the architectural level, design level or code level. The selected objectives specify measurable criteria for evaluating quality characteristics in resultant software systems. Typically, quality objectives are established in a top down manner, which involves identifying a set of high-level quality goals, such as functionality, maintainability, as specified in ISO 9126, subdividing and refining them into more specific attributes (e.g., design decisions, software code) at lower levels. The lowest level attributes can be directly measurable using software metrics. Moreover, quality objectives are selected according to critical quality bottlenecks and target application scenarios. For example, object oriented designs offer properties such as encapsulation, inheritance, and polymorphism. These properties make software systems easier to reuse and maintain. In the scenario of migrating procedural code into object oriented platforms, we aim to achieve two quality objectives namely, reusability and maintainability.

4) *Construct quality models:* quality models provide a representation that depicts the quality refinement process for the selected objectives. Typically, one quality model is used to describe one quality objective. However, models can be interconnected and affect each other. We adopt soft goal interdependency graphs [5] to build software quality models that link a rationale on whether a transformation can achieve a specific non-functional requirement with internal source code attributes that are altered by a given transformation. In general, soft goals are achieved in a top-down manner satisfying the appropriate sub-goals on each step of the process. In this context, changes to code features by transformations in the lower levels of the soft goal hierarchy can be traced up to reflect changes in the root of soft goal interdependency graphs.

5) *Quality measurement:* For the software quality modeled in a soft-goal interdependency graph, a set of metrics can be selected to compute the corresponding source code features that appear as leafs in the soft-goal interdependency graph. The metric results of the leaf nodes indirectly reflect the satisfaction of their direct or indirect parent nodes in the soft-goal interdependency graph. In the context of our research, we are interested in examining a number of product metrics and features that are

related to reusability and maintainability. For example, we aim to achieve high cohesion, and low coupling. These quality goals are considered as sub-goals, and consequently achieve higher-level goals such as reusability and maintainability [4].

3. Quality Driven Software Evolution

Structured programming languages, object orientation, design patterns, and object oriented refactoring exemplify innovative support to improve qualities in the design of software products. Unfortunately, many software developers due to time to market constraints, still adopt a culture of hack and patch regardless of the benefits of such approaches on the successful evolution of software systems. In the context of software evolution, software system should meet the functional requirements, but also need to be tailored to satisfy specific non-functional requirements, such as reliability, performance and reusability for application domains. Although refactoring operations are proposed to improve the design of the code, there is no systematic guidance to improve and enhance the code design towards a specific quality requirement. We believe that incorporating quality constraints is a central aspect of software evolution. To guide the qualities of an evolving software towards desired goals, changes to the software should be understood systematically. Moreover, their impacts should be identified, evaluated and justified.

In our previous research, we have proposed a quality driven transformation framework that allows for existing systems to achieve specific quality goals through a sequence of source code transformations. The original system is first analyzed to detect error prone areas, and determine appropriate strategies for reengineering. To this end, the error pone areas refer to logic artifacts that violate the target software quality requirements. Such logic artifacts can be as simple as source code features, or as abstract as designs and architecture components. For example, in the process of migrating procedural code into object-oriented architecture, the existence of global variables violates the concept of encapsulation. Therefore, the error prone areas must be captured and removed in order to conform to target requirements for the new system. Furthermore, a sequence of transformations should be identified to detect the error prone source code features, and generate the desired target artifacts. To incorporate quality control in the transformation process, each transformation is selected and applied according to its potential impact on the desired qualities in the target system. Specifically, we aim to make explicit the association between source code features and desired software qualities. Moreover, we provide quantitative methods that are used to measure the impacts of each transformation on the desired system properties. The transformation process would terminate either when the highest achievable measured level of quality for the target system has been reached or no transformations can be further applied.

In the context of software development process, source code is constantly changed to meet the evolving requirements. We adopt the proposed quality driven transformation framework to analyze continuous code changes and achieve overall quality improvement towards target quality requirements. We focus on evaluating the quality impact occurred by a sequence of refactoring transformations. We are currently investigating using the CVS repository data to assist in performing quality impact analysis.

4. Software Quality Evaluation

To assess whether the desired goals in the evolved system have been achieved, a set of software metrics and source code features are evaluated. In our research, we conduct software quality evaluation in two steps. In the first step the transformation process is denoted by a state transition model where alternative transformations are assessed on their potential impact to a desired adoption criterion. In the second step, the path or sequence of transformations that has the potential to yield a migrant system with the desired properties is selected and applied.

To select a transformation among various applicable ones, a quantitative approach is taken where system characteristics and metrics that relate to the desired target quality or property are compared when alternative transformations are used. In this context, to assess the impact of a transformation or a sequence of transformations a set of widely accepted object oriented software metrics are used. For example, in the context of refactoring object oriented systems, we aim to achieve high cohesion and low coupling in the newly evolved object oriented systems. In this respect, coupling between classes can be measured using metrics, such as, CBO (Coupling Between Objects), DCC (Direct Class Coupling) and IFBC (Information Flow Between Classes)[6]. Moreover, the cohesion inside a class can be measured using metrics, such as, TCC (Tight Class Cohesion), Coh (Cohesion Measurement) and IFIC (Information Flow Inside Class) [6]. To reflect the overall value of the entire system in terms of one specific metric, we calculate the average value of the each metric value from each class. One single metric cannot reflect the overall quality of a system. To compare the overall quality between alternative systems, we count the number of the metric results which are higher in the optimal system than the others. The goal of the evaluation is to validate that the resultant system generated from the optimal path has the highest qualities in comparison to the

other resultant systems generated from the alternative paths.

5. Conclusion

Quality characteristics are increasingly becoming more and more important at each stage of the software life cycle. If such characteristics are ignored the quality of the evolving system will deteriorate and the system will eventually be halted with possibly a large cost to the organization that owns and operates it. In this paper, we presented issues related to adoption-centric software evolution that provide a framework whereby evolution activities do not occur in a vacuum but aim to produce a target migrant system that possesses specific design and quality properties. As future research, we plan to trace the quality requirements in the evolving system, and establish the association of quality requirements with the source code patterns, alternative designs and architectures. This work is performed in collaboration with IBM Canada Center for Advanced Studies, and is also sponsored by the Consortium for Software Engineering Research.

References

[1] M.M. Lehman, "Programs, life cycles, and laws of software evolution", Proc. IEEE, v. 68, 1980.

[2] Elliot J. Chikofsky and James H. Cross II, “Reverse Engineering and Design Recovery: A Taxonomy, IEEE Software, vol. 7, January, 1990.

[3] Thomas E. Volman, “Software Quality Assessment and Standards”, Computer, Vol. 26, No. 6, June 1993.

[4] Ying Zou, Kostas Kontogiannis, “Migration to Object Oriented Platforms: A State Transformation Approach”, Proceedings of IEEE International Conference on Software Maintenance (ICSM), Montreal, Quebec, Canada, October 2002, pp.530-539.

[5] J. Mylopoulos, L. Chung and B. Nixon, “Representing and Using Nonfunctional Requirement: A Process-Oriented Approach”, IEEE Transactions on Software Engineering, Vol 18 No. 6, June 1992.

[6] Lionel C. Briand, J¨urgen Wst, and Hakim Lounis. Replicated case studies for investigating quality factors in object-oriented designs. Empirical Software Engineering: An International Journal, 6, 2001.

Autonomic Computing - Engineering and Scientific Challenges

Marin Litoiu
Centre for Advanced Studies, IBM Toronto Lab
marin@ca.ibm.com

Abstract

Autonomic computing is emerging as a new software research field, at the confluence of software engineering, systems management, and automatic control. The goal of autonomic computing is to develop software systems that are self-managing, that is, systems that are self-configuring, self-optimizing, self-protecting, and self-healing. This position paper provides an overview of the engineering, scientific and adoption challenges the new research field may face before becoming a widely accepted engineering endeavor.

1. Introduction

Autonomic computing [8][10][17] is about systems that manage themselves. The main rationale behind the concept of autonomic computing is economical: the number and the complexity of computing systems are increasing at a rate that makes their management a challenge, and the cost of ownership a burden. But there is also a technical reason why autonomic computing will prevail: in many complex engineered products, automation is a principal catalyst in ensuring the quality and performance differentiation of those products; it is, therefore, logical to see this happen in software engineered products.

Autonomic computing empowers the computing system with four capabilities: self-configuring, self-optimizing, self-healing, and self-protecting.

Self-configuring systems automate the tasks of installing, integrating, and configuring software. In a world in which a distributed system is made of tens of heterogeneous components, and expert programmers often work for months to install and configure a distributed application, a new approach is needed. Self-configuring components and systems are able to discover each other, dynamically bind and integrate to fulfill global goals defined by the system architect or system administrator.

Self-optimizing systems continuously monitor their own performance and reallocate hardware and software resources as needed to meet the performance requirements with minimal resources.

Self-healing capabilities allow the systems and subsystems to detect malfunctioning, auto-diagnose, identify the faulty components, and recover from any damage. Self-protecting capabilities prevent, detect and correct any accident, hardware failure, or external threat such as a hacker attack or virus.

This paper discusses the challenges in implementing these requirements. We look at engineering and scientific difficulties as well as adoption issues. The rest of the paper is organized as follows. Section 2 enumerates several precursors of autonomic computing and defines the elements of an autonomic element. Section 3 defines the technologies that enable autonomic computing and presents the challenging issues. A possible road ahead for autonomic computing is sketched in Section 4. Section 5 illustrates several of the risks that might prevent the widespread adoption of autonomic computing, and Section 6 presents the concluding remarks.

2. Autonomic Elements

Examples of autonomic computing features already exist. Many of them can be found on the Internet, which contains several self-protecting and self-healing mechanisms [6]. When a link or router fails, an alternative path from source to destination is automatically searched and a new connection is established. If some of the information is lost on the way, the receiver can detect this, and the lost packets are retransmitted and reassembled at their destination. If some of the packets are corrupted on the way, the error detection and correction mechanisms, especially the checksum field of the packets, allow the receiver to restore the data. The Internet also has self-optimization mechanisms: an optimal transmission rate is achieved by the source, destination, and intermediate routers

through a *sliding window protocol*, and *congestion detection and avoidance* algorithms. Self-configuring on the Internet has to do mostly with the addition or removal of nodes or network segments. Once an IP address is obtained and attached to a node, the node becomes part of the Internet without having to worry about bit rates, packet sizes, or the nature of the physical link--radio, satellite, or wire.

Many other software applications manifest autonomic computing characteristics. These include: 1) operating systems with separate execution environments for different programs such that a failure of a process or program does not affect the rest of the processes, and 2) firewalls and demilitarized zones (DMZ) in intranet applications that protect from intrusion. Self-optimization characteristics are present in Internet load balancers and workload managers, which direct the user requests to the least utilized node [15], and in multimedia streaming applications that adapt the transmission qualities of service to cope with congestions in the network. An example of self healing is the garbage collection service provided in many modern programming languages. Self-management characteristics can be found in P2P applications, in grid computing, and more recently in sensor networks.

Compared with the above-mentioned features, autonomic computing has a holistic view of the system and employs feedback on a larger scale. Figure 1 shows the components of an autonomic element, the primitive block that constitutes an autonomic system. A *Managed Element* is a software component (object, process, or system) that performs some functional role. It has *sensors* that allow the *Monitor* to collect data about how the managed element performs its functions. Collected, filtered, and preprocessed data is *analyzed* by the *Autonomic Manager*. Based on the goals and the current state, a plan is devised. The *plan* is *executed* by the manager, which will interact with the managed element through *effectors*.

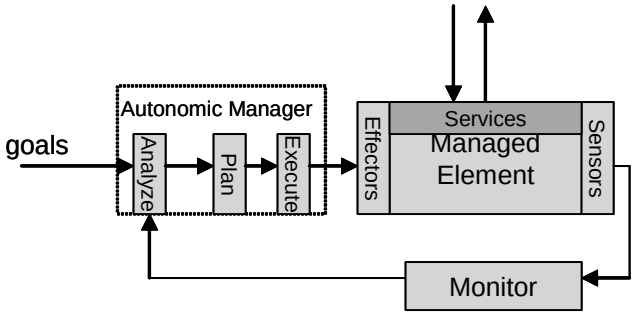


Figure 1. An autonomic element

An autonomic application is a composition of individual autonomic elements, as shown in Figure 2. Alliances between autonomic elements are made and broken dynamically in a P2P or hierarchical manner.

3. Enabling Technologies

The ability to monitor and analyze a managed element, to create a plan of action, and then execute the plan, requires the convergence of several technologies and research fields.

Monitoring requires common data formats to deal with data coming from different sources and to allow integration with many autonomic managers. Several efforts are under way, most notably in the Hyades project [7], which defines a Common Base Event format for traced and logged data from the managed elements. Dynamic active filtering, which selectively chooses the type and number of data sensors, can help reduce the volume of collected data. Reference architectures and dedicated middleware for collecting, transmitting, and storing sensor collected data are equally important. The Astrolable project [16] researches the use of hierarchical distributed databases to store collected data. Middleware for collecting and reliably and efficiently storing and retrieving session states is discussed in [12].

Analysis of the collected data includes data mining, abstract model building, model solving or checking, and prediction. Every aspect of autonomic computing uses its own analysis methods and algorithms. For example, self-optimizing approaches have reported the successful use of both analytical [1] and experimental [5] modeling. Self-healing and self-protecting characteristics might rely on a Law-Governed Interaction [13] mechanism as well as on statistical learning.

Executing a plan, whether or not it pertains to optimization of healing, presumes a flexible software infrastructure. That means that it should be possible to dynamically discover services offered by autonomic elements, estimate the trade-off in using one service over another, bind the services, and test and run them. Many emerging technologies can support the scenario alluded to above. BPEL4WS specifies the behavior of the system in business process terms: activities, data and control flows. Activities are bound to Web services [18] and implemented by autonomic elements. If a service is to be replaced, then, based on the specifications of BPEL4WS, a new service is searched in a universal registry, for example UDDI[20]. Semantic Web[21] facilitates the finding of the appropriate service through Web service ontologies. Another technology that will play a major role in autonomic computing is Open Grid Service

Architectures (OGSA)[19] that envisions the delivery of grid services as Web services.

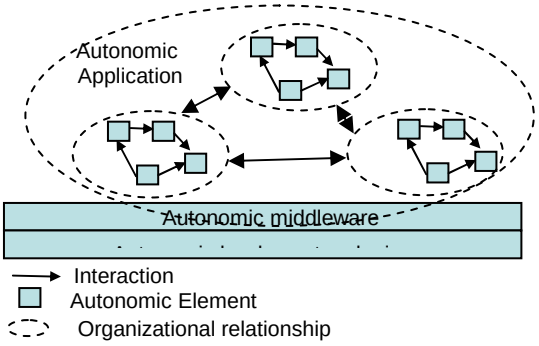


Figure 2. Autonomic applications

Software engineering of autonomic elements and systems is essential to making autonomic computing succeed. Autonomic applications are made of many autonomic elements connected together to accomplish user-defined goals. Languages and methodologies that specify goals have to make sure that: (a) the user has the flexibility and the support to define the desired behavior, and (b) the goals are correctly defined. Agent-oriented software engineering [9] or goal oriented languages [4] can be a starting point. Testing, verifying, deploying, and maintaining autonomic elements and systems will require new advances to support the dynamic nature of the autonomic reconfiguration.

4. The Road Ahead or About Governors

The story of automation in other engineering fields might very well be indicative of what will happen with autonomic computing. At the beginning of the steam engine revolution (1789), James Watt was preoccupied with a simple engineering problem: how to make the engine keep a constant rotation speed, independent of the workload. The solution was a simple but ingenious device, called a fly-ball governor. A fly-ball governor (Figure 3) is a sensor and an actuator at the same time: it has two balls B attached through arms at the end of a spindle. The spindle and the balls rotate with the engine's shaft, and as the shaft rotates faster (because of low workload), the balls are swung further away from the spindle by centrifugal force. The movement of the balls is transmitted to a sliding piece R attached loosely to the spindle. In the end, a throttle-valve closes, adjusting the steam in the engine and decreasing the rotation speed of the shaft.

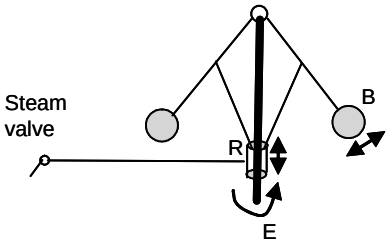


Figure 3. The fly-ball governor

Watt's governor had many drawbacks: it could only be used for one type of engine, it could adjust the speed only for a small range of workloads, it overshot, oscillated, and it was unstable. It took about 100 years to improve the governor, and thousands of governor patents were granted throughout the world in the 19th Century. In 1878, Maxwell published his now-famous paper "On governors," in which he modeled the governor dynamics through differential equations. The paper was a milestone in the history of governors and in that of automatic control because it offered theoretical insights needed to find solutions to a wide range of control problems.

Similar to the governor's history was the introduction of automation in other engineering fields: telephony (electronic negative feedback amplifier), anti-aircraft control (servomechanisms), and aviation (navigation and position control, automatic pilot, automated landing).

It is, therefore, foreseeable that the autonomic computing road will follow the same patterns:

The period of experimental autonomic computing engineering. In the next few years, many autonomic elements and "governors" (a.k.a. autonomic managers) will be created. It is important that best practices are made public to allow researchers to exchange and build on the successful ideas. It is also conceivable that the number of software components equipped with sensors, effectors, and even monitors will increase rapidly. The pace of autonomic managers will be slower, though. In [3], it is reported that at the beginning of the machinery automation, the ratio of instrument to machinery grew rapidly. However, "the majority of the instruments sold were measuring, indicating and recording devices. Towards the end of the period, the sales of controllers started to increase."

The period of scientific autonomic computing. Experimental autonomic computing will solve many

problems but the problems of overshooting, oscillations and stability, and many more that will arise, are hard to solve without a proper mathematical apparatus. In general, the analysis, planning, and execution phases within the autonomic manager are hard to tackle. The automatic control theory [11] used in today's automation solutions is not entirely adequate for the complexity of autonomic computing. Moreover, any of the four autonomic computing capabilities might require specific scientific foundations. Self–optimizing might still rely heavily on the first order principles of the queuing theory, classical optimization, and modern automatic control. Self-configuration, self-healing, and self-protection, however, require a different scientific support: it may incorporate social theories, data mining, learning theory, and artificial intelligence. More importantly, a theory that embraces all four aspects of autonomic computing is required. This theory includes negotiation, conflict resolution, and decision making.

5. Adoption Issues

The road to autonomic computing faces many adoption issues. One issue is the risk of over-automation, the transfer of too many responsibilities to the system. This is also a matter of trust. How do we know the system will perform the way we want, and how will system administrators react? A good story to remember here is the history of cockpit automation [14]. It was believed in the 1970s that the computer would fly the plane and the pilot would monitor and take control only if something went wrong. It was discovered, though, that human beings were terrible at passive monitoring and they quickly got bored, missing things and losing the context. In the 1980s, the philosophy changed and made room for a more human-centered approach, in which the pilot has the central role and uses the automation as needed. It is, therefore, important that we keep a user-centric approach when building autonomic systems.

Complexity, as well as correctness, is a risk that has to be mitigated in order to make the autonomic computing acceptable. Autonomic computing adds another complexity level to the already complex software. Tools and an infrastructure that supports the development and the verification of the automatic elements are needed. But even with future advances in correctness and software verifications, it might not be possible to write error- and bug-free software, and the users can still make mistakes. Recover Oriented Computing (ROC)[22] assumes the worst and focuses on building software that can recover from disasters produced by human operator mistakes or operating systems crashes. Another approach, Internet Scale Operating Systems (ISOS)[2], tries to avoid the loss of data. It develops worldwide distributed systems in which data is split in millions of pieces and redundantly distributed on computers across the globe. The original data is restored on-demand by collecting pieces of data in a P2P approach.

Probably, the most effective way to get autonomic computing adopted by users is by building high-quality automation. The quality of automation can be measured with several metrics. Figure 4 illustrates the behavior of a quality attribute (throughput, for example) of a software system when perturbed by a workload increase or by a failure of one or more software or hardware components. The quality attribute is set by the system administrator to be at a specific level, the horizontal straight line in the figure. The perturbation, which affects the system at the origin of the *time* axis, drops the quality attribute to zero. A good autonomic manager will bring the quality back to normal in a short time (*rise time*) by provisioning the system with the needed resources. Very likely, the system will be over-provisioned, but the *overshooting* must be small since it can mean a high usage of scarce resources. There will be some oscillations around the target value, but after a preferably short time (*settling time*), the quality attribute should stabilize. Optimizing these quality attributes was essential in other engineering fields, and it will be very important to autonomic computing too. In Figure 4, there are also two examples of undesirable unstable systems: one that keeps oscillating, and one that cannot reach the target value.

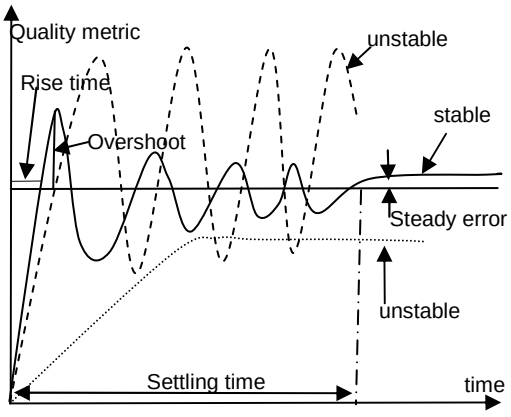


Figure 4. Quality metrics of an autonomic system.

Other qualities that can make an autonomic computing system acceptable and adoptable include robustness (insensitivity to changes in the managed element and environment) and a high tolerance to for perturbations.

6. Conclusions

As in any maturing engineering field, software systems are entering a phase in which automation is a mandatory requirement. This position paper presents an overview of the engineering and scientific challenges that autonomic computing faces. Although the difficulties are many and hard to overcome, we show that there are encouraging signs and that the effort will succeed. There are many new emerging technologies that enable the autonomic computing; there are also lessons to be learned from the automation of other engineering fields. Adoption of the technology will depend on many factors but mainly on the quality of the autonomic systems.

7. References

[1] Abdelzaher T., Shin K., Bhatti N., "Performance Guarantees for Web Server End-Systems: A Control Theoretical Approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, pp. 80-96, January 2002.

[2] Anderson D., Kubiatowicz J., "The Worldwide Computer," Scientific American, pp. 40-47, March 2002.

[3] Benett S., "A Brief History of Automatic Control," IEEE Control Systems, pp. 17-25, 1996.

[4] Bolchini D., Mylopoulos J., "From Task-Oriented to Goal-Oriented Web Requirements Analysis," WISE'03, 2003.

[5] Gandhi N., Hellerstein J., Parekh S. S., Tilbury D., Diao Y, "MIMO Control of an Apache Web Server: Modeling and Controller Design," Proceedings of American Control Conference, May 2002.

[6] Hassan M., Jain R., High Performance TCP/IP Networking, Pearson Prentice Hall, 2004.

[7] Hyades Project, http://www.eclipse.org/hyades/, April 2004.

[8] IBM, "Autonomic Computing: IBM's Perspective on the State of Information Technology, http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf, May, 2004

[9] Jennings N., "On Agent-Based Software Engineering," Artificial Intelligence, No 117, pp. 277-296, 2000.

[10] Kephart J. O., Chess D. M., "The Vision of Automatic Computing," IEEE Computer, 36 (1), pp. 41-52, 2003.

[11] Kuo B., Golnaraghi F., Automatic Control Systems, John Wiley & Sons, 2003

[12] Ling B., Fox A., "A Self-Tuning, Self-Protecting, Self-Healing Session State Management Layer," Proceedings of the IEEE Autonomic Computing Workshop, 2003.

[13] Minsky N., "On the Conditions for Self-Healing in Distributed Software Systems," Proceedings of the IEEE Autonomic Computing Workshop, 2003.

[14] Rudisill M., "Crew/Automation Interaction in Space Transportation Systems: Lessons Learned from the Glass Cockpit," Proceedings of the Human Space Transportation & Exploration Workshop, 2000.

[15]IBM, "Tivoli Provisioning Manager and Tivoli Intelligent Orchestrator," http://publib.boulder.ibm.com/tividd/td/IBMTivoliIntelligentThinkDynamicOrchestrator1.1.html

[16] Van Renesse R., Birman K., Vogels W., "Astrolable: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining," ACM Transactions on Computer Systems, Vol. 21, No 2, pp. 164-206, 2003.

[17] Waldrop M., "Autonomic Computing, the technology of Self-management," http://www.thefutureofcomputing.org/Autonom2.pdf, April 2004.

[18] H. Kreger, "Web Services Conceptual Architecture," http://www-4.ibm.com/software/solutions/webservices/pdf/WSCA.pdf, April 2004.

[19] Foster I. et al., "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration," http://www.globus.org/research/papers/ogsa.pdf, April 2004.

[20] UDDI, "UDDI Technical White Paper," http://www.uddi.org/whitepapers.html, April 2004.

[21] T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web," Scientific American, pp. 28-37, 2001.

[22] Patterson D. et al.," Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies," tech. report CSD-02-1175, Computer Science Dept., Univ. of Calif., Berkeley, March 2002.

The views expressed in this paper are those of the author and not necessarily those of IBM Canada Ltd. or IBM Corporation.

On the Adoption of an Approach to Reengineering Web Application Transactions

Scott Tilley
Department of Computer Sciences
Florida Institute of Technology
stilley@cs.fit.edu

Damiano Distante
Department of Innovation Engineering
University of Lecce, Italy
damiano.distante@unile.it

Abstract

The design, development, and evolution of a large-scale Web site is a very challenging task. For modern Web-based business applications in particular, the lack of a systematic transaction design methodology presents special problems. This paper discusses issues related to the adoption of an approach for reengineering Web application transactions (which are a way for supporting business processes). The approach relies on the recovery of the conceptual model underlying the transaction processes. Potential advantages of adopting this approach, in the context of injecting a more engineering-oriented approach to Web site evolution, are summarized. Several techniques for fostering adoption of the approach are also outlined.

Keywords: *adoption, reengineering, transaction design, Web site evolution, conceptual modeling*

1. Introduction

When the first Web Site Evolution workshop took place in 1999, it was noted that "As Web sites age, they suffer from some of the same afflictions as any complex software system: their structure degrades, maintenance becomes increasingly problematic, and legacy applications and interfaces hinder evolution" [14]. In the intervening five years, the situation has not substantially improved. Indeed, it can be argued that modern Web sites supporting complex business processes are more challenging to evolve in a disciplined manner than ever before.

One approach to mastering the complexity of large-scale Web site evolution is to inject a more engineering-oriented approach to all aspects of the design process. In other words, borrow the best practices of software engineering and adapt them as necessary to this new application domain. One of the central tenets of software engineering is the importance of good design methods and practices. Unfortunately, when it comes to modern Web applications, there is very little "best practice" when it comes to transaction design.

In this context, a Web transaction can be defined as the sequence of activities the user has to fulfill using the Web-based application to accomplish a particular task or to reach a particular goal. It is quite common that Web application transactions are simply emulated as a sequence of navigational steps through the pages of the application [6][7]. This means that the user activities corresponding to each step of the transaction is represented as navigational objects, including (unfortunately) the "Back" button of a browser. This approach, even if workable, often conducts to applications that are poor in quality and show erroneous behaviors also documented in literature [11].

Many e-commerce Web sites that are already deployed could improve their users' experience through site reengineering based on systematic analysis of actual transaction paths, so that users' expectations and site design are more closely aligned. Reengineering the site's transactions from a user's perspective can significantly improve the site's usability.

We have developed an approach to Web site evolution via transaction redesign from a user's perspective [3][4]. The technique currently relies on a human analyst who is a subject matter expert (possibly aided by reverse engineering tools that facilitate the recovery of the as-is transaction model) to guide the reengineering process. In our opinion, the adoption of this approach could be one step towards a more systematic and engineering-oriented approach to Web site evolution.

The next section of the paper summarizes the transaction reengineering approach used to facilitate Web site evolution. Section 3 describes what we see as the primary advantages of adopting the technique for modern Web applications. Section 4 outlines possible methods for fostering adoption of the technique. Finally, Section 5 discusses possible avenues for future work.

2. The Transaction Reengineering Approach

Web site evolution can be achieved using a reengineering approach that relies on extensions to the transaction model portion of the Ubiquitous Web Applications (UWA) design framework [21], and a reverse modeling technique that is used to populate instances of the conceptual model using data gathered from existing Web applications. Extensive details of the extensions to the UWA transaction model, and the reverse modeling technique, are provided in [3][4]; this section summarizes the transaction reengineering approach by providing an overview of the salient extensions and techniques.

2.1 The UWA Design Framework

The UWA is one of the few frameworks that provide a complete design methodology for transaction-oriented Web sites and Web applications that are multi-channel, multi-user and generally context-aware. (In the UWA vernacular, "transactions" represent the way business processes are addressed and implemented in Web-based applications.) Using the UWA can help designers in managing complex Web application by improving communication both within the design team and with external clients. The UWA framework encourages designers to create better-organized design documentation so that future evolution is made easier.

The UWA design framework organizes the process of designing a Web application into four main activities [16]: (1) requirements elicitation [17]; (2) hypermedia and operation design [18]; (3) transaction design [19]; and (4) customization design [20]. Breaking the design of Web applications into several activities and multiple steps in such a manner does not add complexity to the design task (as it may appear to), because it helps in obtaining a separation of concerns and has many known advantages. Among the activities of the UWA framework, hypermedia and operation design may be considered to be "typical" of Web applications design. However, requirements elicitation, transaction design, and customization design are new activities; their definition and integration with the W2000 methodology [10] is one of the main contributions of UWA.

Using the concepts provided by the four UWA design models (and their corresponding notation), a designer can craft the application schema. The schema should take into consideration the overall application requirements, including both typical business requirements and future migration paths. For each design activity, schemas can be specified at two levels of depth and precision: (1) in-the-large, to describe general aspects (sometimes informally), without many details; and (2) and in-the-small, to describe selected aspects in more detail.

2.2 Extensions to the UWA Transaction Model

Since the UWA design framework focuses specifically on transactions in Web applications by means of the Transaction design activity, it represents an excellent platform on which to build. In a UWA-based project, Web application transaction designs are based on two models: the Organization model and the Execution model [22]. The extensions to the UWA transaction model to support transaction reengineering include simplifications and extensions related to the definition of Activity, and enhancements to several aspects of the Organization and Execution models.

2.2.1 Changes to the Definition of Activity

Activities taken into account by the Organization and Execution model of a transaction implementing a process should only be those that are meaningful for the user of the Web-based application; system-related activities and data-centered operations can be deemphasized. This implies that the OperationSet of an activity is no longer considered, mainly because it is primarily related to data level details and to the implementation of a transaction, whereas user-centered transaction reengineering is more concerned with conceptual models.

An Activity's PropertySet is redefined to be more user-oriented through the introduction of a new property (Suspendability), and a tuning of the semantics associated with the previously existing properties. The extended PropertySet set is now Atomicity, Consistency, Isolation, Durability, and Suspendability (ACIDS). An activity a_1 is defined as suspendable if it can be stopped during a session of work and continued in a following one from the exact point where it was left unfinished.

2.2.2 Changes to the Organization Model

The Organization model describes a transaction from a static point of view, modeling the hierarchical organization in terms of Activities and sub-activities in which the Activity can be conceptually decomposed. It also describes the relations among these activities and the PropertySet of each of them. The Organization model is a particular type of UML class diagram [5], in which activities are arranged to form a tree; the main activity is represented by the root of the tree and corresponds to the entire transaction, while Activities and sub-activities are intermediate nodes and its leaves.

Significant changes have been made to the Organization model by dividing the possible relations between an activity a_1 and its sub-activities $a_{1.1}$ .... $a_{1.n}$ into two categories: the Hierarchical Relations and the

Semantic Relations. Hierarchical Relations are the set of "part-of" relations from the Organization model composed of (but not limited to) the relations *Requires*, *Requires One*, and *Optional*. Semantic Relations are the set of relationships that are not a "part-of" type: *Visible*, *Compensates*, and *Can Use*.

These changes to the Organization model provide a better modeling instrument with which transaction reengineering can be accomplished. In particular, the distinction between hierarchical and semantic relations permit the designer to reason about transactions in a manner not possible with the unadorned UWA model. This in turn can lead to improvements in support for the business processes realized by the Web application.

2.2.3 Changes to the Execution Model

The Execution model of a transaction defines the possible execution flow among its Activities and sub-activities. It is a customized version of the UML Activity Diagram [9], usually adopted by the software engineering community to describe behavioral aspects of a system. In the execution model, the sequence of activities is described by UML Finite State Machines, Activities and sub-activities are represented by states (ovals), and execution flow between them is represented by state transition (arcs).

The original Execution model includes both user- and system-design directions for the developer team. Since our focus is more on the former than the latter, several changes have been introduced. Moreover, mixing both kinds of design aspects complicates the task of the designer and results in a model that more difficult for the client to understand. We think the system related aspects should be specified in a different model and a subsequent design phase.

For example, two pseudo-states (commit and rollback) that exist in the original UWA execution model have been removed; positive conclusion of an Activity is now directly derived by the execution flow in the model, while the failure or the voluntary abort of it is modeled by the unique pseudo-state of "Process Aborted" in an Execution model.

Each possible user-permissible transition between activities must be explicitly represented in the model with a transition line between them. The actions that trigger the transition should be specified on the transition line using a simple and extensible labeling mechanism to indicate the category of the transition: A (action invoked by the user); C (condition(s) required for Activity execution); R (result of the execution of an Activity); and S (state associated with the system due to the execution of an Activity). A list of the causes of Activity failure and possible actions the user or the system can take is also maintained.

These changes to the Execution model provide better visibility into the dynamic execution paths the user will experience while completing a specific transaction. By making such paths explicit, improvements in the transaction design can be more easily accomplished. It is also suggested that swimlane diagrams [12] be adopted when it's useful to describe how two or more user types of the application collaborate in the execution and completion of a transaction. However, for such paths to be modeled properly for existing Web sites, they must first be recovered.

2.3 The Reverse Modeling Technique

Given an existing Web site, the goal is to populate an instance of the extended UWA transaction model described above with data from the site's content and structure. The resultant model can then be used to guide reengineering decisions based on objective information concerning the quality attributes of the business process' implementation by the Web-based application. The model can be recreated using a reverse modeling technique that is executed in three steps: (1) formalization of the transactions; (2) creation of the Execution model; and (3) construction of the Organization model. Compared to the way transactions are designed in a forward engineering process, the order in which Execution Model and Organization Model are drawn is inverted. This is because in a reverse design process, it is easier to investigate the dynamics of a transaction first and then, from the results of this analysis, derive the static Organization model.

2.3.1 Formalization of the Transactions

In the first step of the procedure, the user types of the application and their main goals/tasks are formalized. Only goals/tasks that can be defined as "operative" are considered and a transaction is associated with each of them. Overlapping tasks of two or more user types suggests UML swimlanes in the corresponding transaction's Execution model. At the end of this step the list of transactions implemented by the application is obtained.

2.3.2 Creation of the Execution Model

For each of the transactions found with the previous step, the Execution model is created by first performing a high-level analysis of the transaction in order to gain a basic understanding of its activities and execution flow. The transaction is then characterized as "simple" (linear), or "composite" (with two or more alternative execution paths). Each transaction (simple and composite) has an activity of the Execution model (and later of the Organization model) associated with it.

A first draft of the Execution model is created for each transaction by executing it in a straightforward manner, and at the same time, drawing the linear sequence of Activities it is composed of. The model is then refined with deeper analysis of the transaction: all the operations available to the user during the execution of the transaction are invoked, and erroneous or incomplete data are provided in order to model failures states and possible actions the user can undertake. The reverse modeling technique is invoked recursively as needed. Finally, the table that describes the possible failure causes and the corresponding user actions or system invocations is investigated for each of the sub-activities that have been found.

2.3.3 Construction of the Organization Model

Once the Execution model has been obtained for a transaction, the Organization model can be constructed, which will model the transaction from a static point of view. The Execution model is used to determine the set of Activities and sub-activities that the transaction is composed of.

The tree structure of the Organization model is constructed by aggregating sub-activities that are conceptually part of an activity ancestor. Each arc in the tree represents either a hierarchical or semantic relation. To define hierarchical relations, the analyst can refer to the execution flow defined by the Execution model and the relations of composition between transactions and activities found out during its drawing. However, defining the semantic relations still requires direct inspection of the application.

For each Activity and sub-activity, it is necessary to define the value for the ACIDS PropertySet. The analyst is required to refer to the definition given for each of the properties and discover the value to be assigned to each of them through direct inspection using the Web-based application.

3. Advantages of Adopting the Approach

The transaction reengineering approach described in Section 2 represents one step towards the goal of injecting a more engineering-oriented approach to Web site evolution. Three of the most important advantages of adopting the approach are that it is based on established hypermedia design concepts and sound software engineering principles, it works at the conceptual level, and is it focused on supporting the user's point of view. This section briefly discusses each of these advantages.

3.1 A Solid Basis

The transaction reengineering process is grounded in established hypermedia design concepts and sound software engineering principles. The approach is based on two items: extensions to an existing and successful transaction design model (UWA), and a recovery technique that borrows heavily from reverse engineering. Each item itself has firm theoretical background and proven practical value.

3.1.1 The Extended UWA Transaction Model

The choice of the UWA was motivated by several factors, including available documentation and prior experience using UWA, the maturity of the framework itself, and its focus on transaction design. The UWA framework was one of the results of the UWA project [13]. As such, prior experience using the framework is readily available. Moreover, there is extensive documentation that can be used to support any proposed extensions to the framework.

The UWA framework has been tested and proved in several research situations and real-world projects. At the moment it is one of the most suitable frameworks for designing Web applications. As an example, it has been successfully applied to business domains like order management [2] and e-banking [16].

Perhaps most importantly, the UWA framework focuses specifically on transactions. The UWA transaction design methodology represents a first attempt to treat and design transactions in a Web application with a formal and systematic approach. The extensions summarized in Section 2.2 further advance this support.

3.1.2 The Reverse Modeling Technique

Reverse engineering has proven to be a useful method of aiding program understanding. It can help the software engineer master applications that are too large for a purely manual comprehension process. Since understanding is a prerequisite to disciplined evolution, the value of reverse engineering in this context has already been established.

The reverse modeling technique is procedural in nature; it is not (currently) tied to any particular tool. As such, it can be tailored to suit different usage scenarios. For example, where there are human subject-matter experts available to perform the system analysis, the formalization of the transactions can be quickly and accurately accomplished. The creation of the Execution model, and the subsequent construction of the Organization model, can then follow in a natural way.

The result of the reverse modeling process is a recovered "as-is" conceptual model. Since the model has been made explicit (as opposed to hidden in the

application and therefore implicit in the navigation structure), designers can now analyze and evaluate it according to quality attributes such as usability and fulfillment of business requirements in a manner consistent with other engineering artifacts and activities. This in turn aids the reengineering the "as-is" model to create a candidate "to-be" model that may better meets the user's expectations and improves the user's experiences using the Web site.

3.2 A Conceptual Model

The transaction reengineering approach works at a conceptual level, thereby insulating itself from the vagaries of a particular implementation technology. This is particularly important in the context of the rapidly changing Web tool landscape. The final application may be realized using standard Web tools and protocols. However, it is conceivable that it may also be realized for a non-Web platform, for example a special-purpose information kiosk that provides a richer functionality than a simple browser.

By focusing on the conceptual level of the transaction design, the engineering can concentrate on the system's requirements and functional qualities, instead of working directly at the implementation and coding levels. This has long been a goal of many software engineering design techniques. For transaction-oriented Web applications, the same goal is perhaps even more important.

Working at the conceptual level also allows the software engineer to focus on the single most important aspect of the entire application: the user. Instruments such as usability studies (the one presented in [1] is one of possible candidates) and focus groups can be used to guide the redesign of the transaction processes. But this is only possible if the transaction design is explicitly modeled – as it the case using the transaction reengineering approach described in Section 2.

3.3 A User's Perspective

Most Web sites could improve their users' experience through systematic analysis of the transactions implementing business processes, so that users' expectations and site design are more closely aligned. Since transactions represent user interaction with the Web-based application, they necessarily should reflect a user's perspective. Sadly, this is not the case for many e-commerce sites that support complex activities such as online travel reservation.

In contrast, the transaction reengineering approach is closely focused on the user. As stated above, the recovery process results in a transaction design model at a conceptual level. This model is independent of how the site is actually implemented. In this sense, the model is comparable to a mix of a requirements document and a high-level design in a software engineering sense.

The extensions to the UWA framework are centered on making the design process more user-centric. We have deemphasized the system and data requirements portion of the transaction model. Moreover, we decided to renounce most of the rigorous formalism used in the UWA in order to foster the technique's adoption according to our purposes.

4. Techniques to Foster Adoption

The benefits of the transaction reengineering approach won't be fully realized unless the approach enjoys widespread use. There are several techniques that might lend themselves to fostering its adoption. These include increased professionalism through a more engineering-oriented approach to Web site evolution, providing automated tool support (where possible) for the approach (in particular for the reverse modeling technique), and reducing the barrier to using the approach (and the supporting tool if available) by integrating it into existing Web design framework.

4.1 Increased Professionalism

One of the laments of software engineers who examine current Web site construction practices is the lack of discipline and training apparent in many of the people responsible for developing even the most complicated transaction-oriented Web sites. There is still a common misconception that the skills needed to edit an HTML page are somehow sufficient to design a modern Web site. Of course this is not the case, in the same way that knowing how to program would be enough to make someone a professional software engineer.

One of the key areas that could benefit from a more engineering-oriented approach to Web applications is transaction design. This is an area that requires considerable knowledge and expertise, yet is largely ignored in both software engineering and hypermedia design courses. Increased professionalism would mandate a more systematic design methodology, such as the one enabled by the transaction reengineering described in Section 2. In some ways, this would have a positive feedback effect: increased professionalism would direct people to using methods such as those described in this paper to enable the analysis of transaction design, and at the same time use of the methods would improve the professionalism of the resultant Web site. During the next iteration of the cycle, the quality level would increase even as the problem details evolve [8].

4.2 Tool Support

The availability of tools and environments supporting a method and the models it includes as a solution to a class of problems is a key factor that facilitates the adoption of the method and the models themselves. By automating support in key areas of the transaction reengineering, errors could be reduced and efficiency could be increased. Moreover, the ability to codify best practices in a prescriptive tool would enable less experiences users to adopt the approach without the extensive training sometimes required by purely manual solutions.

Tool support for the transaction reengineering could be directed towards the two underlying aspects of the approach: providing support for managing the conceptual models that are based on the extended UWA transaction design, and providing support for the recovery procedure to populate the models based on analysis of existing Web sites. Each of these activities implies a set of requirements that would be quite challenging to realize.

Some of the more important functional characteristics of the proposed tool include:

- Creating the conceptual model of the "as is" transaction design in a (semi-) automated manner;
- Enabling the analyst to reason about the resultant model, and to perform "what if" scenarios based on proposed reengineering of the transaction processes;
- Modifying and/or generating the code to implement the "to be" transactions, replacing the current legacy set, and thereby lowering the cost of future evolution.

There are also a number of nonfunctional requirements for the toolset. For example, in keeping with the conceptual nature and technology-independent focus of the design model, the tool should be platform independent and application software agnostic.

Having an environment able to respond to the above requirements would support the adoption of the transaction reengineering approach because its use would be faster, easier, and more accurate. However, the tool support would still need to be integrated into existing platforms and processes to be most effective.

4.3 Integration with Platforms and Processes

As stated in [15], "One of the reasons why research tools often remain lab orphans is that it is so difficult for third parties to adopt the solution and make efficient use of it in their own work." The availability of tools to support the transaction reengineering process would partially address this issue by codifying some of the more tedious and error-prone tasks involved in the reverse modeling procedure. However, tools by themselves are not sufficient; the tools must be used to effective.

A proven technique to facilitate adoption is to ensure ease of integration. That is, strive to construct the tool in such a manner that it is easily integrated into existing software and hardware platforms. Similarly, design the technique so that it is easily integrated into existing engineering processes. These goals are easier to state than they are to achieve. But recognizing their importance is an important first step.

For the transaction reengineering procedure in particular, every effort should be made to integrate the approach into existing best practice. This means, for example, making the use of recovered conceptual models as a basis for deliberations concerning transaction design modifications the norm rather than the exception. Increased professionalism will help make this happen, as will automated tool support. But ultimately, people will only adopt a technique if they see significant benefit in its use that far outweigh the costs associated with learning and using the technique in the first place. Hopefully the benefits accrued from adopting the approach to reengineering Web application transactions in Section 2 meet this key requirement.

5. Summary

This paper discussed issues related to the adoption of an approach for reengineering Web application transactions. In our context, a Web transaction is defined as the sequences of activities the user has to fulfill using the Web-based application to accomplish a particular task or to reach a particular goal. Web application transactions are a way to implement and support business processes in Web applications.

The transaction reengineering approach relies of recovery of the conceptual model underlying the transaction processes. The reverse modeling technique is relatively easy to understand and apply systematically. The designer is provided with a sequence of clear steps to be carried out and a set of well-defined concepts to refer to. This allows the analyst to draw from the application and effectively represent with the models a lot of information experienced by the user and worthy of attention from their point of view.

Potential advantages of adopting this approach, in the context of injecting a more engineering-oriented approach to Web site evolution, were also summarized. The advantages rely on the fact that the approach has a solid basis in established hypermedia design principles and

software engineering concepts, it functions at the conceptual level, and it focuses on the user's perspective.

Several techniques for fostering the technique's adoption were also outlined. The techniques included an increase in professionalism on the part of the designer, tool support that would automate some of the more onerous aspects of the recovery procedure and that would enable powerful analysis of the resulting models, and integration of the tool(s) into existing platforms and processes.

The approach can be applied to many common Web site evolution scenarios. For example:

- It can be used to redocument the implementation of the transactions of an existing Web site;
- It can be used to compare the design of the transactions with the requirements provided for the application and the structure of the business processes they rely on, and suggest corrections and improvements to the design;
- It can aid the detection and removal of erroneous behaviors, misalignments with the application requirements, and shortcomings of the current implementation of the transactions.
- It can be used to have a view on the transaction according to the user's perspective and to restructure it in order to improve the user experiences.

All the above-summarized observations make the approach desirable for the designer charged to document and evolve the transaction-related portion of a legacy Web application. It provides the analyst with a clear and systematic way to recover, represent, and reengineer the transactions implemented in a Web application. The adoption of the technique would make the Web transaction reengineering process repeatable, its results understandable, and its methods shareable among knowledgeable designers and the community at large.

References

[1] Bolchini, D., Triacca, L., Speroni M., "MiLE: a Reuse-Oriented Usability Evaluation Method for the Web." In *Proc. International Conference on Human-Computer Interaction* (HCII 2003: Crete, 2003

[2] D. Distante, V. Perrone, M. Bochicchio, "Migrating to the Web a Legacy Application: The Sinfor Project." *Proceedings of the Fourth International Workshop on Web Site Evolution* (WSE 2002: October 2, 2002; Montréal, Canada), pp. 85-88. Los Aamitos, CA: IEEE CS Press.

[3] Distante, D.; Parveen, T.; and Tilley, S. "Towards a Technique for Reverse Engineering Web Transactions from a User's Perspective." To appear in *Proceedings of the 12th IEEE International Workshop on Program Comprehension* (IWPC 2004: June 24-26, 2004; Bari, Italy). Los Alamitos, CA: IEEE CS Press, June 2004.

[4] Distante, D.; Tilley, S. and Parveen, T. "Web Site Evolution via Process Restructuring from a User's Perspective". Submitted to *The 20th IEEE International Workshop on Software Maintenance* (ICSM 2004: Sept. 11-17, 2004; Chicago, IL). April 2004.

[5] G. Booch, J. Rumbugh, I. Jacobson, *The Unified Modeling Language User Guide*. (Rational Corporation Software), Addison-Wesley.

[6] G. Rossi, H. Schmid, F. Lyardet, "Engineering Business Processes in Web Applications: Modeling and Navigation Issues." *Proceedings of the 3rd International Workshop on Web Oriented Software Technology* (IWWOST 2003: Oviedo, Spain, 2003).

[7] H. Schmid, G. Rossi, "Modeling and Designing Processes in E-Commerce Applications." *IEEE Internet Computing*, January/February 2004.

[8] Huang, S.; Tilley, S.**;** and Zhou, Z. "On the Yin and Yang of Academic Research and Industrial Practice." *Proceedings of the 3rd International Workshop on Adoption-Centric Software Engineering* (ACSE 2003: May 9, 2003; Portland, OR), pp. 19-22. Published as CMU/SEI-2003-SR-004. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, June 2003.

[9] J. Bellows, "Activity Diagrams and Operation Architecture", CBD-HQ White paper, www.cbd-hq.com, Jan. 2000.

[10] L. Baresi, F. Garzotto, P. Paolini, "Extending UML for Modeling Web Applications." *Proceedings of the 34th Annual Hawaii International Conference on System Sciences* (HICSS-34: Honolulu, HI, 2001). Los Alamitos, CA: IEEE CS Press.

[11] L. Baresi, G. Denaro, L. Mainetti, P. Paolini, "Assertions to Better Specify the Amazon Bug." *14th ACM International Conference on Software Engineering and Knowledge Engineering* (SEKE 2002: Ischia, Italy, 2002).

[12] Object Management Group (OMG), "Unified Language Modeling Specification, version 1.5", www.omg.org, Mar. 2003.

[13] The Ubiquitous Web Applications Project. Online at www.uwaproject.org.

[14] Tilley, S. (editor). *Proceedings of the 1st International Workshop on Web Site Evolution* (WSE 1999). Available online at http://www.websiteevolution.org/1999/.

[15] Tilley, S.**;** Huang, S.; and Payne, T. "On the Challenges of Adopting ROTS Software." *Proceedings of the 3rd International Workshop on Adoption-Centric Software Engineering* (ACSE 2003: May 9, 2003; Portland, OR), pp. 3-6. Published as CMU/SEI-2003-SR-004. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, June 2003.

[16] UWA (Ubiquitous Web Applications) Project, "Deliverable D3 Requirements Investigation for Bank121 pilot application", http://www.uwaproject.org, 2001.

[17] UWA (Ubiquitous Web Applications) Project, "Deliverable D6: Requirements Elicitation: Model, Notation and Tool Architecture", www.uwaproject.org, 2001.

[18] UWA (Ubiquitous Web Applications) Project, “Deliverable D7: Hypermedia and Operation design: model and tool architecture”, www.uwaproject.org, 2001.

[19] UWA (Ubiquitous Web Applications) Project, “Deliverable D8: Transaction design”, www.uwaproject.org, 2001.

[20] UWA (Ubiquitous Web Applications) Project, “Deliverable D9: Customization Design Model, Notation and Tool Architecture”, www.uwaproject.org, 2001.

[21] UWA (Ubiquitous Web Applications) Project, “The UWA approach to modeling Ubiquitous Web Application”, www.uwaproject.org, 2001.

[22] UWA (Ubiquitous Web Applications) Project. “The UWA approach to modeling Ubiquitous Web Application.” Online at www.uwaproject.org (2001).

Testing Challenges in Adoption of Component-Based Software

Ladan Tahvildari
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada, N2L 3G1
ltahvild@swen.uwaterloo.ca

Abstract

The adoption of components in development of complex software systems can surely have various benefits. Their testing, however, is still one of the open issues in software engineering. On the other hand, building high quality and reusable components is very important for component-based software development projects. This means that the testability of software components is one of the important factors determining the quality of components and their adoption as extensions to commonly used office suites and middle-ware platforms. To develop such high quality components, we need to answer concerning component testing and component testability. This paper shares our thoughts and understanding of component testability, and discusses and identifies the challenges and issues concerning to the testing of evolving component-based software systems.

1 Introduction

Component-Based Software Engineering (CBSE) focuses on building large software systems by integrating previously existing software components. By enhancing the flexibility and maintainability of systems, this approach can potentially be used to reduce software development costs, assemble systems rapidly, and reduce the maintenance burden associated with the support and upgrade of large systems.

Since components are intended to be reused across various products and product-families, possibly in different environment, components must be tested adequately. Such high-quality components can be extended and leveraged to provide cognitive support to develop software systems. Although there are many published articles addressing the issues in building component based programs, very few papers address the problems and challenges in testing and maintenance of software components that can be adopted and deployed in industry [6, 16, 17, 18, 20].

With the advances in the software component technology, people have begun to realize that the quality of component-based software products depends on the quality of software components and the effectiveness of software testing processes [2, 5, 14, 19]. As pointed out by Elaine J. Weyuker [20], we need new methods to test and maintain software components to make them highly reliable and reusable if we plan to adopt and deploy them in diverse software projects, products, and environments. By presenting the state-of-the-art in component-based software testing, this paper discusses an in-depth understanding of the current issues, challenges, needs, and solutions for this critical area.

This paper is organized as follows. Section 2 discusses various characteristics of testability in evolving components. Section 3 highlights several research challenges on building and testing of evolving components testability. Section 4 identifies open problems to test evolving component-based software systems. Finally, Section 4 explains our conclusions.

2 Characteristics in Testing Components

A software component is defined as "a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties as Commercial-off-the-shelf(COTS)" [15]. One of the greatest problems with the component technology is fault isolation of individual components in the system and coming up with an efficient test strategies for the integrated modules that use these third party components (COTS).

There are different perspectives of component testability in component engineering, including *component observability*, *component traceability*, *component controllability*, and *component understandability* [2] which are elaborated further in the following sections.

2.1 Component Observability

This perspective of component testability defines the ease with which a component can be observed in terms of its operational behaviors, input parameters and outputs [5]. The design and definition of a component interface thus plays a major role in determining the component's observability.

The technique proposed by Roy S. Freedman [5] to check a component's interfaces can be used to evaluate how easy to observe its operations and outputs corresponding to its inputs. In the practice of component engineering, it was observed that the *component traceability* is another very important factor that affects the component observability.

2.2 Component Traceability

This is the capacity of the component to track the status of its attributes and behavior. The former is called *Behavior Traceability* where the component facilitates the tracking of its internal and external behaviors and the latter is called *Trace Controllability* which is the ability of the component to facilitate the customization of its tracking functions [6].

The component traces can be classified into five types [7]: i) operational trace that records the interactions of component operations, such as function invocations, ii) performance trace that records the performance data and benchmarks for each function of a component in a given platform and environment, iii) state trace that tracks the object states or data states in a component, iv) event trace that records the events and sequences occurred in a component, and v) error trace that records the error messages generated by a component.

2.3 Component Controllability

This indicates how easy is to control a component on its inputs/outputs, operations and behaviors [5]. Controllability of a component is considered in three aspects: i) behavior control that has something to do with the controllability of its behaviors and output data responding to its operations and input data, ii) feature customization that refers to the built-in capability of supporting customization and configuration of its internal functional features, and iii) installation and deployment that refers to the control capability on component deployment.

We believe component trace controllability is very useful in component debugging, component integration, and system testing. In addition, component testers and customers expect software component vendors to generate components with test controllability to support acceptance testing and unit testing in a stand-alone mode. Component test controllability refers to a component's capability of retrieving and exercising component tests. Unfortunately, most current component vendors do not provide components with this capability due to the lack of research results on how to design and develop testable components.

2.4 Component Understandability

This shows how much component information is provided and how well it is presented. Presentation of component documentation is the first factor. For a component, there are two sets of documents. The first set is written for users. It includes component user manual, user reference manual, and component application interface specifications. The other set is written for component engineers, including component analysis and design specifications, component testing and maintenance documents. The second factor to component understandability is the presentation of component program resources, including component source code and its supporting elements, such as its installation code, and test drivers. The final factor is the presentation of component quality information, including component acceptance test plan and test suites, component test metrics and quality report.

Most current third-party components did provide users with component user manual and application interface specifications. However, only some of them provide user reference manual, and most of them do not provide any quality information. Although it seems reasonable for component vendors to hide detailed test information and problem information, customers will expect them to provide quality information, acceptance test plan, and even test suites for components in the near future.

3 Research Challenges

Testing conducted in development and use of a component can be considered from two distinct perspectives [8],namely *component provider* and *component user*. The component provider corresponds to the role of the developer of a component and the component user to that of a client of the component provider, thus to that of the developer of a system using the component. The component provider and component user need to exchange various types of information during the development of the component itself and also during the development of a system using the component. However, exchange of such information can be limited due to various reasons and both the component provider and component user can face a lack of information.

Limited exchange of information among the component provider and component user is to our opinion the main reason why testing of components is a research problem of its own and needs to be considered in particular. Therefore, the testability of an evolving component-based system highly

depends on the testability of involved components and their integration. There are several concerns on building and testing of evolving components which are elaborated in the following sections.

3.1 Building Reusable Component Tests

Considering the evolution and adoption of software components, we must pay attention to the reuse of component tests. The primary key to the reuse of component tests is to develop some systematic methods and tools to set up reusable component test suites to manage and store various component test resources, including test cases, test data, and test scripts.

In the current engineering practice, software development teams use an ad-hoc approach to creating component test suites through a test management tool. Since existing tools usually depend on different test information formats, repository technologies, database schema, and test access interfaces, it is difficult for engineers to deal with diverse software components with a consistent test suite technology. This problem affects the reuse of component tests in the component acceptance testing and component integration.

To solve this problem, there are two alternatives. The first is to create a new test suite technology for components with plug-in-and-test techniques. With this technology, engineers are able to construct a test suite for any component, and perform component tests using a plug-in-and-test technique. Clearly, it is necessary for us to standardize software component test suites, including test information formats, test database schema, test access interfaces, and to define and develop new plug-in-test techniques to support component unit testing at the unit level.

The other alternative approach is to create component tests inside components, known as build-in tests. Unlike the first approach, where component tests created and maintained in a test suite outside of a component, this approach creates component tests inside components. Clearly, it simplifies component testing and reduces the component test cost at the customer side if a friendly test-operation interface is available. To execute the built-in component tests, we need extra function facilities to perform test execution, test reporting, and test result checking. Therefore, there is a need to standardize test access interfaces supporting the interactions among components, test suites, and built-in tests.

3.2 Constructing Testable Components

An ideal testable software component is not only deployable and executable, but also testable with the support of standardized components test facilities. Unlike normal components, testable components have the following features. Testable components must be traceable. As defined in Section 2, traceable components are ones constructed with a built-in tracking mechanism for monitoring various component behaviors in a systematic manner.

Testable components must have a set of built-in interfaces to interact with a set of well-defined testing facilities. The interfaces include i) a test s et-up interface, which interacts with a component test suite to select and set up black box tests, ii) a test execution interface, which interacts with a component test driver or test execution tool to exercise component functions with a given test, and iii) a test report interface, which interacts with test reporting facility to check and record test results.

Although testable components have their distinct functional features, data and interfaces, they must have a well-defined test architecture model and built-in test interfaces to support their interactions to component test suites and a component test-bed. Testable components with built-in tests must use a standardized mechanism to enclose the built-in tests. With this mechanism, we can access and exercise the built-in tests in a consistent way.

There are some questions regarding to design of testable components. The first question is how to design and define the common architecture and test interfaces for testable components that can be adopted in a new environment. The next question is how to generate testable components in a systematic way. The final question is how to control and minimize program overheads and resources for supporting tests of testable components that can be deployed for future use.

3.3 Building a Generic and Reusable Test Bed

In general, a program test execution environment consists of several supporting functions: test retrieval, test execution, test result checking, and test report. Clearly, a component test environment must include the similar supporting functions. The primary challenge is how to come out a new component test-bed technology that is applicable to diverse components developed in different languages and technologies.

To achieve this goal, we need standardize the following interfaces between components and its test bed. These interfaces are : i) test execution interface between components and a test bed which supports a test bed to interact with components to support test execution, test result checking, and test reporting, ii) test information access interface between a test bed and component test suites which supports a test bed to interact with test suites to retrieve test information, and iii) interaction interface between a test bed and a component test driver or a component test stub.

3.4 Constructing Component Test Drivers and Stubs

Now, in the real world practice, engineers use ad-hoc approaches to developing module-specific or product specific test drivers and stubs based on the given requirements and design specifications. The major drawback of this approach is that the generated test drivers and stubs are only useful to a specific project (or product). It is clear that the traditional approach causes a higher cost on the construction of component test drivers and stubs.

In the component engineering paradigm, components might have the customization function to allow engineers to customize them according to the given requirements. This suggests that the traditional way to construct component test drivers and stubs is very expensive and inefficient to cope with diverse software components and their customizable functions.

We need new systematic methods to construct test drivers and stubs for diverse components and various customizations. The essential issue is how to generate reusable, configurable (or customizable), manageable test drivers and stubs for a component. Component test drivers must be script-based programs that only exercise its black-box functions. There are two groups. The first group includes function-specific test drivers, and each of them exercises a specific incoming function or operation of a component. The second group contains scenario specific test drivers, and each of them exercises a specific sequence of black-box operations (or functions) of a component.

Component test stubs are needed in the construction of component frameworks. Each test stub simulates a black-box function and/or behavior of a component. There are two general approaches to generating test stubs. The first approach is model-based, in which component stubs are developed to simulate a component, in a black box view, using a formal model, such as a finite state machine, a decision table, or a message request-and-response table. Its major advantage is model reuse and systematic stub generation.

To cope with component customization, we need a tool to help us change and customize an existing model, and generate a new model. The second approach is operational script-based, in which test stubs are constructed to simulate a specific functional behavior of a component in a black-box view. The major advantage of this approach is the flexibility and reusability of test stubs during the evolution process of components. However, the challenge is how to generate these function-specific test stubs in a systematic manner.

4 Emerging Trends and Open Problems

This section presents open problems based on i) the various component-testing strategies as suggested both by the academic research and the industrial research world, and ii) the research challenges in adoption component-based systems that discussed in Section 3.

One factor distinguishes issues that are pertinent in the two perspectives, namely *component provider* and *component user*, is the availability of the component source code. The component providers have access to the source code, whereas the component users typically do not. One type of software for which the source code is usually not available is commercial off-the-shelf software (COTS). Although there are no regulations imposed on developers of COTS, to standardize development and reduce costs, many critical applications are requiring the use of these systems [11]. The lack of availability of the source code of the components limits the testing that the component user can perform.

Researchers have extended existing testing techniques for use by component providers. For example, Doong and Frankl describe techniques based on algebraic specifications [4], Murphy and colleagues describe their experiences with cluster and class testing [12], and Kung and colleagues present techniques based on object states [10]. Other researchers have extended code-based approaches for use by component providers for testing individual components. For example, Harrold and Rothermel present a method that computes definition use pairs for use in class testing [9]. These definition use pairs can be contained entirely in one method or can consist of a definition in one method that reaches a use in another method. Buy and colleagues present a similar approach that uses symbolic evaluation to generate sequences of method calls that will cause the definition use pairs to be executed [1].

Researchers have considered ways that component users can test systems that are constructed from components. Rosenblum proposes a theory for test adequacy of component-based software [13, 14]. His work extends Weyuker's set of axioms that formalize the notion of test adequacy [20], and provides a way to test the component from each sub domain in the program that uses it. Devanbu and Stubblebine present an approach that uses cryptographic techniques to help component users verify coverage of components without requiring the component developer to disclose intellectual property [3].

With additional research in these areas, we can expect efficient techniques and tools that will help component users test their applications more effectively. We need to understand and develop effective techniques for testing various aspects of the components, including security, dependability, and safety; these qualities are especially important given the explosion of web-based systems. These techniques can provide information about the testing that will increase the confidence of developers who use the components in their applications.

We need to identify the types of testing information about a component that a component user needs for testing applications that use the component. For example, a developer may want to measure coverage of the parts of the component that his/her application uses. To do this, the component must be able to react to inputs provided by the application, and record the coverage provided by those inputs. For another example, a component user may want to test only the integration of the component with his/her application. To do this, the component user must be able to identify couplings between his/her application and the component.

We need to develop techniques for representing and computing the types of testing information that a component user needs. Existing component standards, such as COM and JavaBeans, supply information about a component that is packaged with the component. Likewise, standards for representing testing information about a component, along with efficient techniques for computing and storing this information, could be developed. For example, coverage information for use in code-based testing or coupling information for use in integration testing could be stored with the component; or techniques for generating the information could be developed by the component provider and made accessible through the component interface.

Finally, we need to develop techniques that use the information provided with the component for testing the application. These techniques will enable the component user to effectively and efficiently test her application with the component.

5 Conclusions

Research for testing evolving component-based systems is still an open problem. We still do not have appropriate methods, techniques and tools supporting both the component provider and component user in testing a component. The overview given in this article has shown the existing approaches in this area. This article focused on the approaches which are applicable by both the component user and the component provider. The author hopes that this article opens discussion started to gain a consensus concerning the problems and open issues in testing evolving component-based software systems.

References

[1] U. Buy and O. A. Issues in testing distributed component-based systems. In *Proceedings of the First International ICSE Workshop on Testing Distributed Component-Based Systems*, May 1999.

[2] W. T. Councill. Third-party testing and the quality of software components. *IEEE Software*, 16(4):55–57, July/August 1999.

[3] P. T. Devanbu and S. G. Stubblebine. Cryptographic verification of test coverage claims. *IEEE Transactions on Software Engineering*, 26(2):178–192, February 2000.

[4] R.-K. Doong and P. G. Frankl. The astoot approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, April 1994.

[5] R. S. Freedman. Testability of software components. *IEEE Transactions on Software Engineering*, 17(6):553–564, June 1991.

[6] J. Gao, H.-S. J. Tsao, and Y. Wu. *Testing and Quality Assurance for Component-Based Software*. Artech House, 2003.

[7] J. Gao, E. Zhu, S. Shim, and C. Lee. Monitoring software components and component-based software. In *Proceedings of the 24th Annual International Computer Software and Applications Conference (COMPSAC)*, pages 403–412, October 2000.

[8] M. J. Harrold. Testing: A roadmap. In *The Future of Software Engineering (special volume of the Proceedings of the International Conference on Software Engineering (ICSE))*, pages 63–72, June 2000.

[9] M. J. Harrold and R. G. Performing dataflow testing on classes. In *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 154–163, December 1994.

[10] D. Kung, P. Hsia, and G. Jerry. *Object-Oriented Software Testing*. IEEE Computer Society Press, 1998.

[11] G. McGraw and J. Viega. Why commercial-off-the-shelf (cots) software increases security risks. In *Proceedings of the First International ICSE Workshop on Testing Distributed Component-Based Systems*, May 1999.

[12] G. Murphy, P. Townsend, and P. Wong. Experiences with cluster and class testing. *Communications of the ACM*, 37(9):39–47, 1994.

[13] A. Orso, M. J. Harrold, and D. Rosenblum. Component metadata for software engineering tasks. In *Proceedings of the 2nd International Workshop on Engineering Distributed Objects (EDO))*, pages 129–144, November 2000.

[14] D. S. Rosenblum. Adequate testing of component-based software. Technical Report UCI-ICS-97-34, Department of Information and Computer Science, University of California, Irvine, August 1997.

[15] C. Szyperski. *Component Software- Beyond Object Oriented Programming*. Addison Wesley, 1997.

[16] L. Tahvildari and A. Singh. Software bugs. In J. G. Webster, editor, *Encyclopedia of Electrical and Electronics Engineering*, volume 19, pages 445–465. John Wiley & Sons, July 1999.

[17] J. M. Voas. Certifying off-the-shelf-components. *IEEE Computer*, 31(6):53–59, June 1998.

[18] J. M. Voas. Maintaining component-based systems. *IEEE Software*, 15(4):22–27, July/August 1998.

[19] Y. Wang, G. King, and H. Wickburg. A method for built-in tests in component-based software maintenance. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 186–189, March 1999.

[20] E. J. Weyuker. Testing component-based software: A cautionary tale. *IEEE Software*, 15(5):54–59, September/October 1998.

Toward a Unified Model for Requirements Engineering

Brian Berenbach
Siemens Corporate Research, Inc.
Princeton, NJ 08540 USA
Brian.Berenbach@siemens.com

Abstract

One of the problem areas in requirements engineering has been the integration of functional and non-functional requirements and use cases. Current practice is to partition functional and non-functional requirements such that they are often defined by different teams. Functional requirements are defined by writing text-based use cases or, less frequently, creating a business model, then walking through the use cases, and extracting (often in a haphazard fashion) detailed requirements.

I believe that it is possible by taking advantage of the extensibility of modern CASE tools to create a single, homogeneous UML use case model that seamlessly and harmoniously connects use cases, functional requirements and non-functional requirements. Using such a model the facilities of UML are leveraged and traceable requirements sets can be extracted.

1. Introduction

The typical process by which requirements are defined on large projects has not changed for several years (Figure 2). Teams of analysts and subject matter experts define functional requirements. Architects and other experts define non-functional requirements. If the resultant set of requirements is large, traceability, readability and verification can be difficult.

There are significant problems inherent in the above approach. Functional and non-functional requirements are defined by different individuals or teams. The separation of the requirements effort usually involves different subject matter experts, and often involves different tool sets. Difficulties are built in to this approach.

Because different subject matter experts are used on different teams, they often don't review each other's work. This means that:

- Functional requirements discovered during architectural sessions may not get communicated to the requirements team and
- Non-functional requirements discovered during modeling or use case writing or reviews may not get communicated to the architectural team.

In order to have an effective risk mitigation process (and to be CMM/CMMI compliant) it is necessary to have full traceability. When the requirements are elicited using different teams, tools, and processes, tracing between requirements can be problematic.

For example, an architectural review exposes the requirement that a certain type of device will be needed in the future. How will this requirement be traced to functional requirements? One common solution is to put all the requirements into a single repository (e.g. DOORS, Caliber, Requisite Pro, etc.) and then, post discovery, create traces between the requirements. This approach breaks down with large databases as it becomes increasingly difficult to identify all the necessary links.

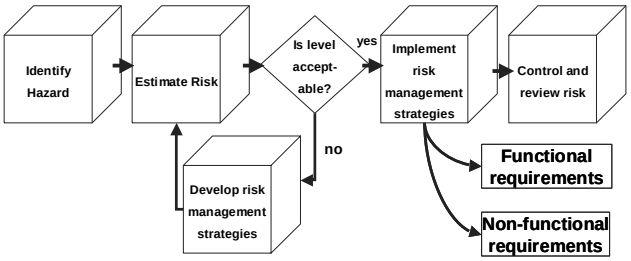


Figure 1 Hazard Analysis Process

2. Global Analysis

Global analysis [2][3] is a high-level design technique, done during the early stages of software architecture design for determining design and project strategies.

Global analysis considers factors that influence the design of the product line, grouped into categories of organizational, technological, and product influences. Analysis of the influencing factors results in a set of design strategies that are used to guide a product line architecture design. The strategies are realized with functional and non functional requirements.

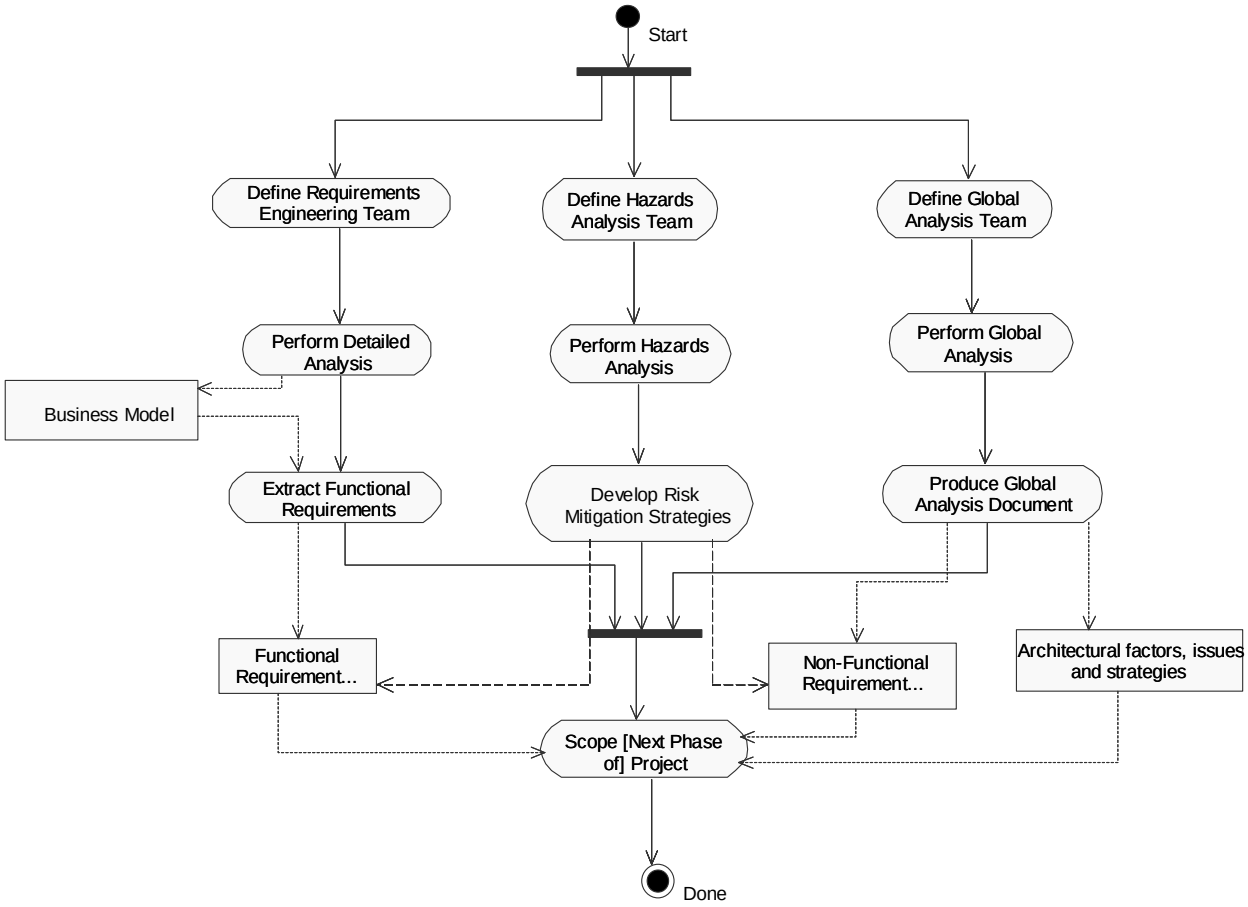


Figure 2 Typical Requirements Capture Process

3. Hazard Analysis

Hazard analysis is an area that presents additional challenges to the requirements analyst. By hazard, I do not mean the risks associated with the project, rather the physical hazards to the end user of the software. An example would be the potential hazard of a person receiving an excessive dose of X-rays when getting a dental checkup. The embedded software that controls the X-ray machine has an identifiable hazard. The risk that this hazard may occur must then be quantified (high, medium, low), and risk mitigation strategies must be identified and implemented. The resultant mitigation strategy may result in the generation of functional and non-functional requirements (see Figure 1). How are these relationships to be identified and maintained. Figure 3 illustrates the difficulty of creating a back trace from a requirement to the hazard that it derives from.

Another difficulty associated with diverse, simultaneous efforts (functional requirements elicitation, non-functional requirements elicitation and hazards analysis) is that of synchronizing requirements with their respective use cases. A large business model can contain several hundred use cases. If the model is well formed, it will not be difficult to navigate. Nevertheless, post-analysis, connecting requirements back to their associated use-cases can be a difficult, time-consuming task.

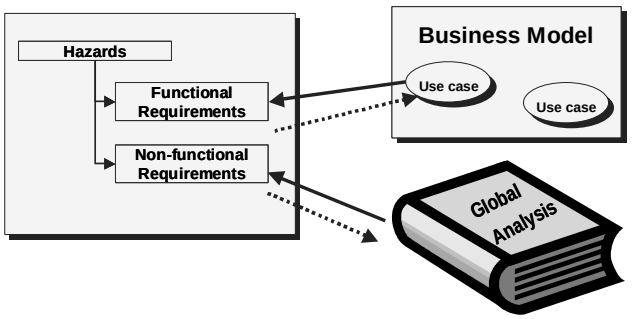


Figure 3 Tracing Requirements To Their Source

To summarize, a project may have up to three disparate teams creating requirements using different processes and tools. As the project moves forward, the artifacts get larger, and the effort to trace requirements back to their origins becomes increasingly difficult (Figure 3).

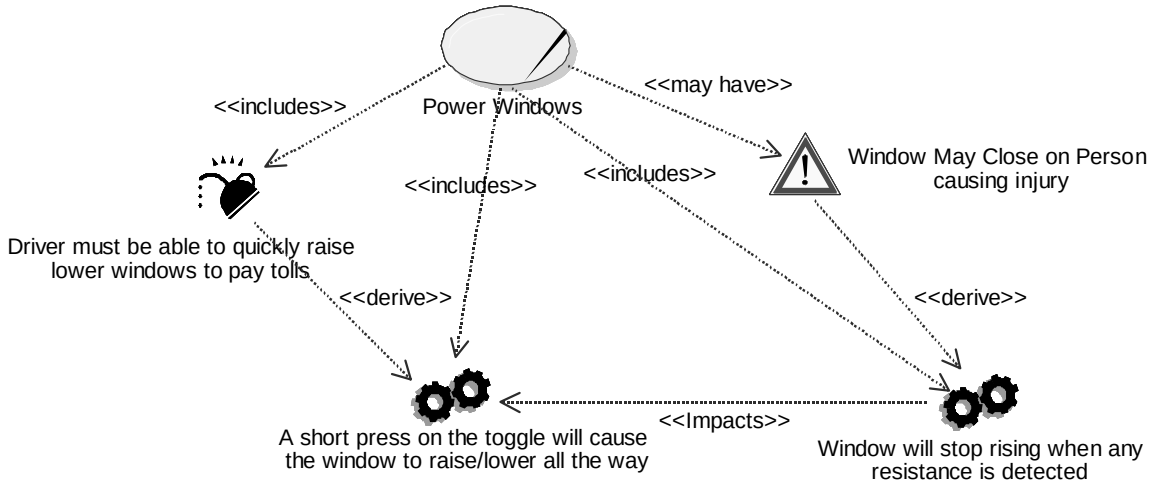


Figure 4 Use Case and associated hazard and requirements

4. A Unified Approach to Requirements Engineering

A new approach is needed that enables multiple teams working on different aspects of requirements on a project to use the same tools and formalisms. Intrinsic support for tracing would be very helpful.

What is proposed in this position paper is such an approach: an extension of UML to support the integration of hazard and requirements analysis and the binding of the resultant exposed requirements to their respective use cases, with enabling tool support.

I have chosen the use case diagram as the starting point for a unified modeling approach, with the UML extensions indicated below.

Hazard Symbol

The specific symbols are not as important as what they represent. For the hazard symbol, I chose the universally accepted hazard symbol. That was the easy one.

Functional Requirement Symbol (RQT)

For functional requirements, I picked the meshed gears sometimes used to show process.

Non-Functional Requirement Symbol (NFRQT)

A colleague, Dr. Andre Bondi, suggested the use of an oilcan for non-functional requirements. If functional requirements are related to process, then the non-functional requirements represent the enablers (e.g. lubricant) for the process.

5. Creating A Unified Business Model

Once the symbols are defined, the next issue is that of relationships; e.g., do the relationships make sense? What are the relationships between use cases, hazards, functional requirements and non-functional requirements?

I have attempted a definition of the relationship stereotypes, shown in Table 1 below. This certainly does not preclude the definition of other relationships; this was a first attempt.

Table 1 Suggested Relationship Stereotypes

From/To	Use Case	RQT	NFRQT	Hazard
Use Case	communicates	includes	includes	may have
RQT	included	impacts	impacts	derived from
NFRQT	included	impacts	impacts	derived from
Hazard	may result from	derives	derives	associated with

Applying the above relationships to Figure 4, I can interpret the diagram as follows: "Rolling up power windows may have an associated hazard that the window can close on a person causing injury. The hazard is mitigated by adding (derives) the requirement to the use case that a window will stop rising as soon as any resistance is detected. This requirement will impact another power windows requirement that a short press on the toggle will completely raise/lower the window. In addition, the driver window must lower quickly enough to enable the driver to pay a toll promptly."

6. Benefits of a Unified Model

I have shown elsewhere that requirements can be programmatically extracted from a well-formed UML model [1]. The model is essentially a directed graph, with requirements, hazards and use cases as vertices, and their relationships as edges. Moreover, the diagrams are just views into the model.

Some advantages of a unified, model-driven approach to requirements elicitation include:

- Simplified Requirement Reuse (Figure 5)
- Stereotyping of requirements dependencies on a per-project basis; e.g., a stereotype could be "release 1" or "mandatory".
- Visualization of requirement inheritance

Non-functional and functional requirements are integrated in a single model, and readily visible on use case diagrams.

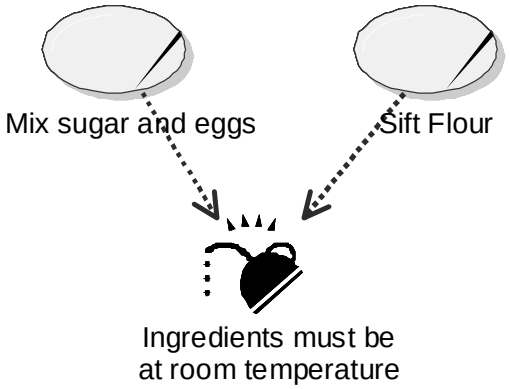


Figure 5 Visualizing Requirement Reuse

7. Integrating CASE Tools with Requirements Repositories.

Commercial repositories tend to present requirements in a tabular fashion. A "spread-sheet" approach is valuable when performing costing exercises, defining a release plan, etc. However, when initially defining requirements, both functional and non-functional, this approach tends to be inadequate, and fosters the process separation between hazards analysis, and functional and non-functional requirements analysis tasks.

With current processes, coupling use cases to requirements and hazards becomes an "afterthought", leading to problems when the repositories and models are large. Additional difficulties arise when attempting queries such as:

- "Is every concrete use case in my model associated with one or more requirements?"
- "Is every requirement in my repository associated with one or more use cases?"
- "Which use cases are impacted by hazards?"

The tight coupling of a requirements repository and CASE tool (Figure 6) might mitigate these problems, especially if the CASE tool can be used as a repository "front end".

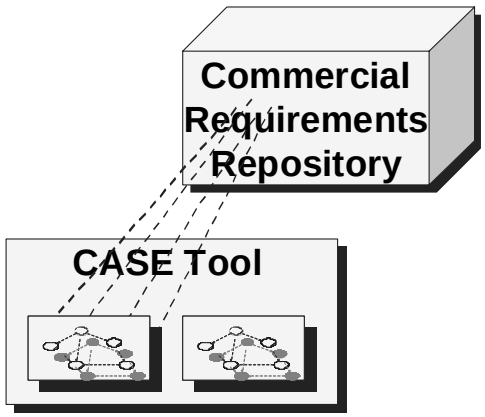


Figure 6 Integration of CASE and Repository Tools

8. Summary and Conclusions

I have found it relatively easy to extend commercial CASE tools to include symbols and relationships for requirements and hazards. Moreover by extending the Use Case Diagram, requirements inheritance relationships also become feasible.

Clearly, it works! Next steps include a pilot project at Siemens to determine the efficacy of this approach: is it scalable, ergonomic, and resilient?

If possible, a CASE tool and/or repository vendor will be found to work with Siemens to create a seamless integration of use cases, requirements and hazards, enabling a modern, unified, model driven approach to requirements engineering.

9. References

[1] B. Berenbach, "The Automated Extraction Of Requirements From UML Models, *Eleventh IEEE International Symposium on Requirements Engineering (RE'03)*, Monterey Bay, Ca, September 2003, pp. 287-288.

[2] Nord, R.L.; Paulish, D.J.; Soni, D.; Hofmeister, C.; Effective software architecture design: from global analysis to UML descriptions; *Proc., 23rd Int'l. Conference on Software Engineering. ICSE 2001*. Toronto, Ont., Canada; 12-19 May 2001; pp. 741-2.

[3] C. Hofmeister, R. Nord, and D. Soni, Applied Software Architecture, 2000, Addison-Wesley, Reading, MA.

Domain Modelling for Adopting Software Components Based Development for Product Families

Muthu Ramachandran* and Pat Allen
School of Computing
Beckett Park Campus
Leeds Metropolitan University
LEEDS LS6 3QS, UK
Email: m.ramachandran@leedsmet.ac.uk

Abstract

This paper provides an industrial application of the domain modelling techniques for adopting component based approach to software development. We will illustrate one of the main domain modelling technique known as FODA for identifying and adopting components for reuse. This paper also provided an example object-oriented classification for the Teletext subsystem of a TV domain. We have also adopted component based approach to product line development and product family for which a model has also been presented in this paper.

1. Introduction

Domain analysis requires software engineering, knowledge engineering, systems analysis techniques, data modeling techniques, architecture engineering, design and modeling techniques, advanced software design techniques (program comprehension, design for reuse and program/product families, generic component design, OO development). What is new in domain analysis? The novelty of domain analysis is the integration of known techniques to create an infrastructure that maximises software reuse at low cost when developing software for restricted classes of applications.

Various approaches to domain analysis currently exist [4 & 7]. Each technique focuses on increasing the understanding of the domain by capturing the information in formal model(s). In this paper we have selected two such approaches to domain analysis (FODA[1] and Prieto-Diaz [2]) that are well known (commonly adopted/applied in practice). These two selected approaches discuss domain analysis methods whereas the other approaches have mainly concentrated on the process. Schafer et al. [7] discusses eight different but similar approaches and provides a comparison table on their merits and demerits.

The Feature-Oriented Domain Analysis (FODA) was developed at the Software Engineering Institute (SEI). FODA defines a process for domain analysis and establishes a specific product for later use [1]. Three basic phases characterise the FODA process:

- Context Analysis: defining the extent (or bounds) of a domain for analysis. It is an activity which is a combination of a technical assessment and a business, or cost/benefit evaluation. Other approaches refer to this activity as selecting, scoping the domain, identifying domain boundaries, and market analysis. The objective is to acquire a better understanding of the investment required to analyse the class of applications and the potential pay-off in reuse. It also involves analysing the variability of external conditions.
- Domain Modelling: providing a description of the problem space in the domain that is addressed by software.
- Architecture Modelling: creating the software architecture(s) that implement solutions to the problems in the domain.

Pros & cons: This model provides specific guidelines on identifying the scope of the domain and feature-oriented approach to domain analysis. However, it fails to provide a detailed approach on how to capture domain knowledge and what the outcome domain model should provide. This model fails to provide an infrastructure for the domain analysis process.

* This work was carried out when the author was with Philips Research Labs, Redhill, UK.

Prieto-Diaz [2] depicts the process of domain analysis by means of a context diagram as shown in Figure 1, which shows the inputs, outputs, controls, and mechanisms of domain analysis. Input information is obtained from existing systems and includes source code, documentation, designs, user manuals, and test plans. Also domain knowledge is input, along with requirements for current and future systems. The personnel involved perform their work with the guidance of domain analysis methods and management procedures. The domain analyst co-ordinates the analysis process. The domain expert supports the input (acquisition) phase, and the domain engineer supports the output (encapsulation) phase. The standards output may include software development methods and procedures, coding standards, management policies, and library maintenance procedures.

There are two major challenging issues to be considered when adopting software components:

1. adopting to components based development itself 9changing the culture and training addressing all classical ssues
2. adopting to product line. That is to say continous improvement by using components and monitoring benefits regularly.

This paper aim to address both issues by providing clear guidelines and models.

2. Adopting Component Based Software Development

There have been a number of works on reuse based product development and product-line approach to software development [21-23]. In general systems are composed of components (software and hardware inclusively). Therefore, component-based systems are comprised of multiple software components that:

- are ready *'off-the-shelf'*, whether from a commercial source (COTS) or re-used from another system,
- have significant aggregate functionality and complexity,
- are self-contained and possibly execute independently,
- preferably used 'as-is' rather than modified,
- must be integrated with other components to achieve required system functionality,
- use component architectures to build resilient architectures

Examples of components based systems can be drawn from many domains, including computer-aided software engineering (CASE), engineering design and manufacturing (CAD/CAM), office automation, workflow management, command and control, and many others. Figure 2 illustrates a model for component-based software development [21]. As shown in this diagram (Figure 2) the model consists of five stages Off-the-shelf components, Qualified components, Adapted components, Assembled components, and Updated components.

The development of embedded systems from components can be considered to be primarily as assembly and integration process. Briefly we describe each key steps as follows:

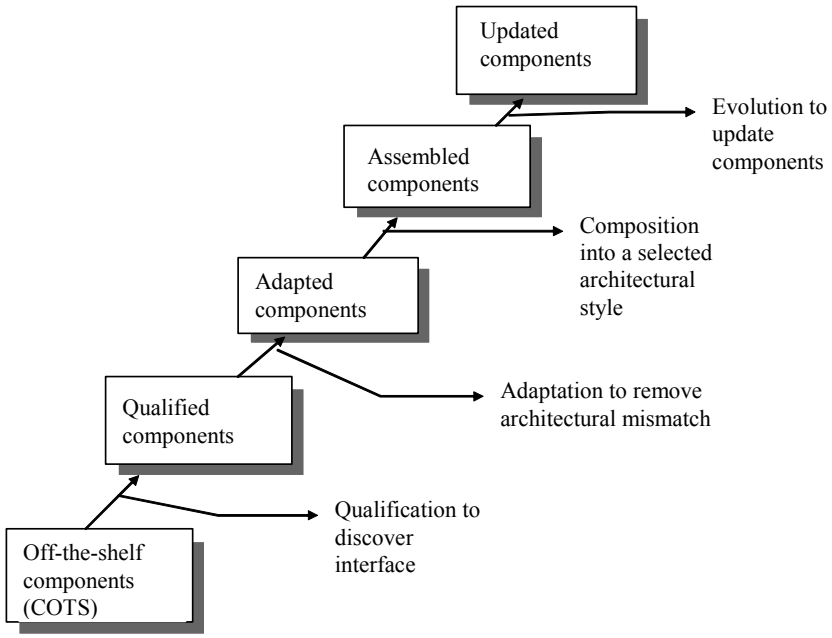


Figure 2 A Model for Component Based Development

- *Off-the-shelf components* have emerged to support product-line approach where identifying and developing reusable components, and interface design play a key role supporting reuse over the product-line. This is an essential first step for starting up a product-line approach.
- *Qualified components* have discovered interfaces so that possible sources of conflict and overlap have been identified. This is a partial discovery: only those interfaces important to effective component assembly and evolution are identified. How to qualify/certify a component as a potential

candidate for reuse remains an active research topic.

- *Adapted components* have been amended to address potential sources of conflict when trying to match for a new set of requirements.
- *Assembled components* have been integrated into an architectural infrastructure. This infrastructure will support component assembly and co-ordination, and differentiates architectural assembly from ad hoc "glue".
- *Updated components* have been replaced by newer versions, or by different components with similar behaviour and interfaces. Often this requires wrappers to be re-written, and for well-defined component interfaces to reduce the extensive testing needed to ensure operation of unchanged components is not adversely effected.

However, existing practices have not been able to solve some of the issues emerged when developing reusable components:

- how to identify and develop reusable components that can guarantee a return-on-investment (ROI)
- what makes a component more reusable for embedded systems
- what kind of a business model for this new approach
- who will be responsible for making go/no-go decision
- how to we conduct evaluation, validation, verification, and testing (VVT) of components

Starting up a reuse programme is as hard as reusing them. This is known as *Domain Engineering* where reusable assets are identified and developed. The process involves finding commonalties among systems to identify components that can be applied to many systems, and to identify product families that are positioned to take fullest advantage of those components. The notion of how to reuse and maintain those components is known as *Application Engineering*. The third notion is automated generation of reusable frameworks and architectures from a higher level specification is known as *Generation Engineering*. Software Architecture plays a major role to structure systems so that they can be built from reusable components, evolved quickly, and analysed reliably.

3. Component Evaluation in Practice

In reality we can evaluate reusable components after they have been reused in products for some time. There by collecting data such as the feel and experience of the reusers, failure rate, effort to modify, reuse benefits in terms of productivity, etc. Some of the approaches to evaluation are,

- obtaining objective data on the components and its technology by documenting case studies at other organisations;
- gathering subjective opinions and experiences with the introduction of this new technology by attending exhibitions, industrial practices, seminars, conferences, and by conducting interviews or by sending out questionnaires to vendors and users of the technology;
- conducting focused experiments to mitigate high-risk aspects of a new technology;
- demonstrating the feasibility of a new technology by executing a pilot project;
- comparing a new technology to existing practices by conducting a shadow project and examining the results of both approaches;
- phased exposure to a new technology by initiating demonstrator projects within a small part of the organisation. Similar approach can be followed when during Evaluation and VVT.

Sometimes an organisation focuses on one of

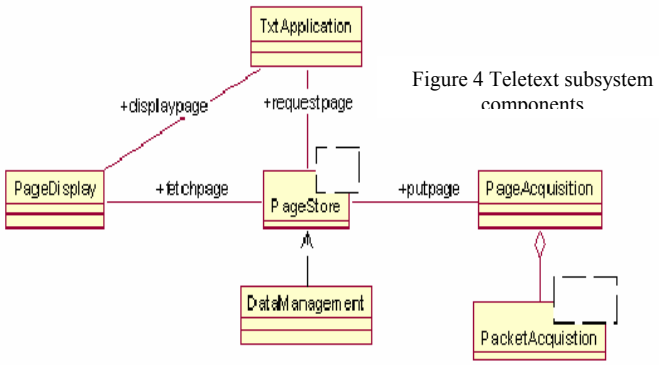


Figure 4 Teletext subsystem components

these approaches, while at other times some combination of the approaches is employed. Regardless of the approaches used we find that what is missing is a well-developed conceptual framework for technology evaluation that allows the results of the evaluation to be considered in terms of what this new technology adds to the existing technology bases.

4. Case study: a TV product family

TV systems are getting more software intensive due to demands for various services and the emergence of digital broadcasting has impacted this consumer market rapidly. A TV product line inherits from general display systems and then splits into two major categories such as analogue and digital systems. In our work we only consider the digital TV family. The digital TV splits further into networked systems and standalone (home) systems. The networked systems are often used in a hotel and museum, etc. A standalone digital TV system consists of more than 100 variations and which grows rapidly based on the market competition. These product variations include plasma TV, Flat TV, normal TV, DVD-comi, VHS-combi, and further more we also need to include TV sizes such as 14", 21", 28", 32" etc. The software needs to be configured for each product variations. The user interface varies quite significantly. The UI management needs to consider languages, country, buttons, icons, and messages for internationalisation. Figure 3 illustrates an example of a product family and product line for consumer products.

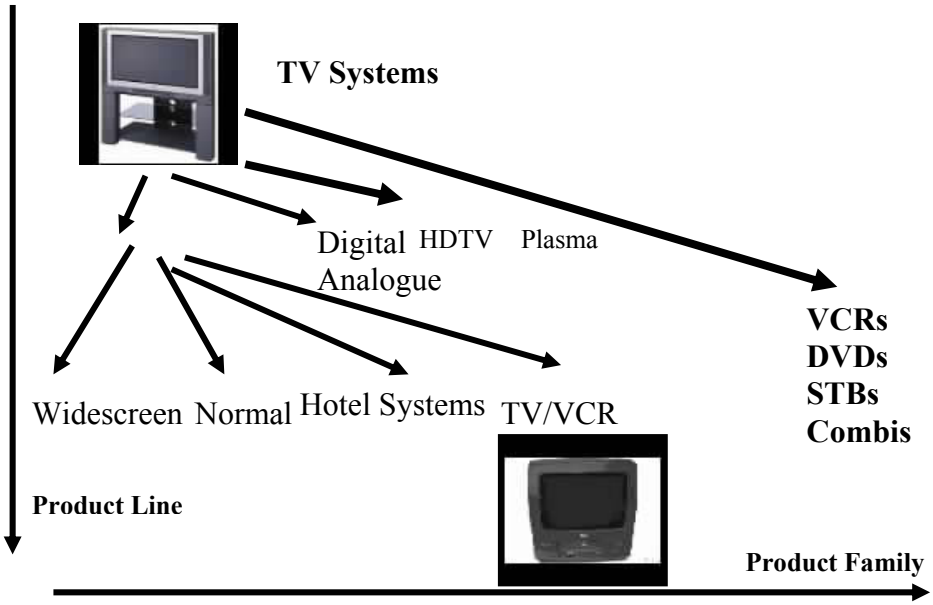


Figure 3 Product Family and Product Line for consumer products

5. Information model for teletext subsystem

In this section we provide object modelling for the teletext subsystem which is one of the major complex software application in a TV product. Figure 4 shows an example of a component based teletext subsystem in UML notation. The classes such as PageStore, PacketAcquisition are modelled as parameterised classes so that they can be re-instantiated for various situations (this is an example of providing a rationale for making a design decision). These are the design decisions that must be analysed with domain experts. The discussion with domain experts will ensure the following:

- The component has a potential for reuse hence justifying the investment on developing and maintaining a component;
- The component must be a generic, parameterised, subsystems, etc., to assess certain design decision with support from its reuse potential;
- Experts/designers need to concentrate on choosing a class type, relationships, and parameterisation when designing reusable components.

We propose only three types of reusable component for a product line.

- Generic components (maximum reuse vs effort required, needs a balance), shown as white box in the above diagram.
- Block components (similar to unix pipeline concept, maximum reuse potential))
- Interface components (medium reuse benefits).

6. A Model for Adopting Components for Product Family

Most of the current approaches provides some interesting insights but are lacking in many ways when it comes to supporting large scale domain modelling for product line reuse. Therefore we propose the following domain modelling process for adopting component based software development to product family as shown in Figure 5.

- Identify product line family tree. Product 1.1, Product 1.2, Product 1.3, etc.
- Develop at least one component for each branch in the commonality tree.
- Identify commonality features and variable features in new product versions. Draw a ven diagram illustrating shared frames
- Classify commonality features (this is also know as domain classification which is rather difficult to predict before hand if no family product exists)
- Identify and develop three classes of reusable components, *generics*, *building blocks*, and *fixed interfaces*.

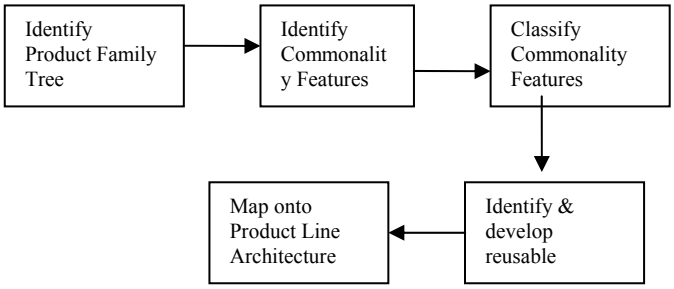


Figure 5 Domain Modeling for Product Line

7. Conclusions

Adopting components based approach to productivity is a challenging area of research. In practice this involves not only choosing a technology for adopting software development but includes training and cultural change. Domain analysis will hep to understand the domain well for adoption to a new product. The notion of reuse infrastructure and roles involved when conducting the domain analysis process, are essential ingredients for a successful component-based development initiative. This paper also provided an example object-oriented classification for the teletext subsystem of a TV domain. This paper has taken application view of the approaches to domain modelling for adopting components successfully.

References

[1] Kang, K.; Cohen, S.; Hess, J.; Novak, W., Peterson, A. Feature-oriented domain analysis feasibility study, technical report, CMU/SEI-90-TR-21.

[2] Prieto-Diaz, R., Domain analysis: an introduction, ACM Software Eng. Notes 15(2), 47-54, April 1990.

[3] Nilson, Roslyn; Kogut, Paul; & Jackelen, George Component Provider's and Tool Developer's Handbook Central Archive for Reusable Defense Software (**CARDS**). STARS Informal Technical Report STARS-VC-B017/001/00. Unisys Corporation, March 1994.

[4] Hooper, J.W, Chester, R. O, Software Reuse, Plenum Publications, 1991.

[5] Jacobson, I., M. Griss, M., Jonsson, P., Software reuse: architecture, process and organisation for business success, Addison-Wesley, 1997. Oriented Domain Analysis to the Army Movement Control Domain (CMU/SEI-91-TR-28, ADA256590). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1992.

[6] Kang, Kyo C.; Cohen, Sholom G.; Hess, James A.; Novak, William E.; & Peterson, A. Spencer. Feature-Oriented Domain analysis, IEEE Software, 2001

[7] Analysis (FODA) Feasibility Study (CMU/SEI-90-TR-21, ADA235785). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1990.

[8] Rumbaugh, James, et al. Object-Oriented Modeling and Design. Englewood Cliffs, NJ: Prentice Hall, 1991.

[9] Atkinson, C. et al. Component-Based Product Line Engineering with UML, The KobrA Method, Tutorial Session, UML 2000, November 2000, York, UK.

[10] Karlsson, E. A.(Ed.)., Software reuse: a holistic approach, John Wiley 1995.

[11] Schafer, W., Prieto-Diaz, R., Matsumoto, M. Software Reusability, Ellis Horwood, 1994.

[12] America, P., Domain Engineering Survey, http://nlww.natlab.research.philips.com:8080/research/swa_group/america/domain/domain.htm#Literature

[13] Foreman, J., Product Line Based Software Development- Significant Results, Future Challenges. Software Technology Conference, Salt Lake City, UT, April 23, 1996.

[14] Weiss, D., Commonality Analysis: A Systematic Process for Defining Families. Second International Workshop on Development and Evolution of Software Architectures for Product Families, February 1998.

[15] Simos, M., Organization Domain Modeling (ODM): Formalizing the Core Domain Modeling Life Cycle. SIGSOFT Software Engineering Notes, Special Issue on the 1995 Symposium on Software Reusability, Aug 1995.

[16] Jean-Marc DeBaud: PuLSE Product Lines for Software Systems. Slides for tutorial at ERW'98, November 4, 1998.

[17] Jean-Marc DeBaud: PaceLine: A Software Product Line Engineering Process. Slides for a tutorial, February 1999.

[18] Desmond D'Souza and Alan Wills: Objects, Components, and Frameworks with UML: The Catalysis Approach. Addison-Wesley, 1999.

[19] Philippe Kruchten: The Rational Unified Process, an introduction. Addison-Wesley, 1999.

[20] Balzer, R. et al. 3rd international workshop on Adoption-Centric Software Engineering ACSE 2003, Proceedings of the 25th international conference on Software engineering, May 2003.

[21] A. Brown, Component-Based Software Engineering, IEEE Computer Society Press, 1996.

[22] Microsoft COM/DCOM models.

[23] Cheesman, J. and Daniels, J. (2000) UML Components, Addison Wesley.

Using Components to Build Software Engineering Tools

Holger M. Kienle
University of Victoria
Victoria, Canada
kienle@cs.uvic.ca

Marin Litoiu*
IBM Toronto Lab
Toronto, Canada
marin@ca.ibm.com

Hausi A. Müller
University of Victoria
Victoria, Canada
hausi@cs.uvic.ca

Abstract

Traditionally, software engineering research tools have been developed from scratch without much reuse. Software components offer now a different way of tool-building, which replaces hand-crafted code with pre-packaged functionality. Tools based on components have unique characteristics for developers as well as tool users. This paper describes three common approaches to tool-building with components and evaluates the approaches from a tool-builder's perspective.

1. Introduction

> "*Programs these days are like any other assemblage—films, language, music, art, architecture, writing, academic papers even—a careful collection of preexisting and new components.*"
> – Biddle, Martin, and Noble [5]

Tool-building is a fundamental part of software engineering research, resulting in a significant expense in terms of both cost and labor. Many researchers develop tool prototypes, which serve to prove the feasibility of certain concepts and as the basis for user studies to further enhance the tools. These prototypes are often iteratively refined, sometimes resulting in industrial-strength applications. Thus, tool building is a necessary, pervasive, resource-intensive activity within the software engineering community.

As a consequence, researchers have tried to reduce the cost of tool-building by reusing existing code. A promising approach is the use of preexisting software components to implement software engineering functionality. However, the use of components changes the development of tools and has unique benefits and drawbacks. Research tool development should focus on the novel features of a tool. Components can help because they provide a baseline environment, which albeit often trivial to accomplish, requires significant effort to code from scratch. On the other hand, using components means that developers have less control over their code base and face a steep learning curve before getting productive.

The paper is organized as follows: Section 2 introduces three component architectures that enable tool-building via programmatic customization. Examples are given for each architecture, illustrating how software engineering tools can be realized. Section 3 evaluates the benefits and drawbacks that the discussed component architectures have for tool-building. Section 4 draws some conclusions.

2. Tool-Building with Components

We take a broad view of what constitutes software components, defining them as "building blocks from which different software systems can be composed" [8]. Thus, a component can be a commercial off-the-shelf (COTS) product, an integrated development environment (IDE), or any other program.

A prerequisite of tool-building with components is that the target component offers sophisticated customization mechanisms. Support for customization can be divided into non-programmatic and programmatic customization mechanisms. Non-programmatic customization is accomplished, for example, by editing parameters in startup and configuration files or with direct manipulation at the GUI level. Programmatic customization involves some form of scripting or programming language that allows the modification and extension of the component's behavior.

Programmatic customization of a component can be accomplished via an Application Programming Interface (API) (cf. Section 2.1) and/or a scripting language (cf. Section 2.2). The Eclipse framework extends the API approach with a multi-level architecture (cf. Section 2.3).

2.1. Components with APIs

Figure 1 shows the architecture of a component offering an API. Commercial components are typically black-box.

*The paper represents the views of the author rather than of IBM.

In this case extensions are constrained by the functionality of the API. An example of such a component is IBM's **Montana** [15]. Montana is an extensible C++ IDE with an incremental C++ compiler. It is a black-box component with a sophisticated C++ API, which allows compiler-related extensions such as style checkers and stub generators. Extensions are described declaratively in special text files and are loaded dynamically.

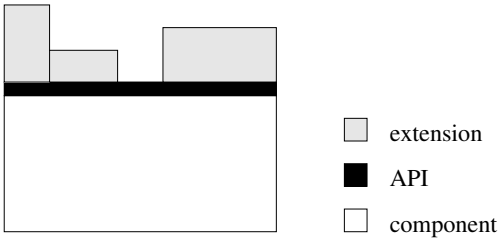


Figure 1. Component with API

Part of the API are observer extensions, which notify clients at distinguished execution points during the compilation process. Since extensions are written in C++ and have write access to the internals, they can crash Montana. Montana assumes that writers of extensions "know what they are doing" [28].

An example of an extension is the C++ fact extractor of the Rigi reverse engineering tool [21]. Using a component such as Montana can greatly reduce the development effort. The Rigi extractor uses the Montana API to generate facts about the compiled C++ source in Rigi Standard Format (RSF) as a side-effect of the compilation process.

Another example of a component with an API is Adobe **FrameMaker**. The FrameMaker Developer's Kit has a C API that makes it possible to implement clients that programmatically extend the GUI and manipulate documents (e.g., for grammar checkers and report generators) [2]. FrameMaker also notifies clients via callbacks of user actions (e.g., opening a file or entering a character).

The Desert software development environment uses the FrameMaker API to realize a specialized editor for source code editing and architecture documentation [26]. The editor uses a wide variety of FrameMaker's functionality, such as syntax highlighting with fonts and colors, graphic insets, and hypertext links. Desert shows that using "FrameMaker for program editing is practical" [25].

2.2. Components with Scripting

Components can also offer a scripting language to simplify programmatic customization. Sometimes, scripting is offered as an alternative to a traditional API, which requires a compiled language such as C. Figure 2 shows a component that has both an API and scripting. The scripting language provides an alternative to directly program the API.

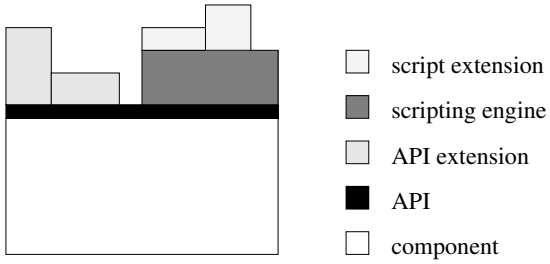


Figure 2. Component with Scripting

A prominent example of such a component is the emacs text editor, which can be customized by programming in Emacs Lisp. Similarly, the AutoCAD system can be customized with AutoLisp [12]. An early commercial product that allowed users to create customized applications was HyperCard, making "everybody a programmer" [14].

An example of a highly customizable research tool is **Rigi** [35] [34]. Rigi is an interactive, visual tool designed for program understanding and software re-documentation. The core of Rigi is a generic graph editor (written in C/C++) enhanced with functionality for reverse engineering tasks. The Rigi graph editor allows customization by exposing its graph model with Tcl. Rigi's user interface is fully scripted in Tk, which makes it easy to modify and extend. Programmers can script customizations with the Rigi Command Language, which is a library of around 400 Tcl/Tk procedures.

Because of its high level of customization support, Rigi has been used as the baseline environment for other reverse engineering tools (e.g., Bauhaus [4], Dali [16], and Shimba [31]). Scripts have also been used to provide domain-specific reverse-engineering analyses (e.g. for Web site understanding [22] [17], metrics [32]), and detection of Java code smells [38].

The most prominent example of a scriptable component is Microsoft Office. The components (e.g., PowerPoint, Excel, and Visio) expose their object models via COM. Customization can be done in COM-aware languages such as Visual Basic (for Applications) and C++.

The Rigi group at the University of Victoria has customized **Visio** to provide similar functionality as the original Rigi tool [41]. This tool, called REVisio, uses Visio's diagramming engine to visualize and manipulate reverse engineering graphs. RENotes uses Visual Basic scripts to add GUI elements and to render a Rigi graph with Visio's built-in shapes and connectors. Visio's object model encompasses almost all data in Visio, including canvas and shape information, menus and dialogs. Thus, clients are able to modify existing elements in any way, as well as add

new ones, or simply access information about the canvas' current contents.

Another example of a popular scriptable component is **Lotus Notes**. Notes exposes its internal state with the Domino Object Model [36]. This model allows programmatic access to and manipulation of its databases and application services. It has been implemented for a broad range of languages, including Lotus Script, Visual Basic, JavaScript, and Java. Similar to ReVisio, the RENotes tool customizes Notes to graft reverse engineering technology on top [20].

2.3. Eclipse

Eclipse has been conceived as a framework for creating programming environments and various languages are already supported (e.g., Java, C/C++, and Cobol). But the framework is also general enough to serve as a baseline environment to graft software-engineering functionality on top.

Eclipse has a flexible extension mechanism based on plug-ins, extensions, and extension points. A new contribution to Eclipse consists of a number of plug-ins. Plug-ins are written in Java and have a declarative specification in XML (a.k.a. manifest file), which states the extension points that the plug-in offers and the extensions (typically provided by other plug-ins) that it plugs into.

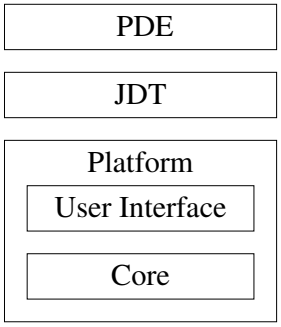


Figure 3. Layered Architecture of Eclipse

The Eclipse framework has three main layers: The Plug-In Development Environment (PDE), the Java Development Tools (JDT), and the Platform (see Figure 3). The Platform defines the common, application-neutral infrastructure of the IDE, consisting of concepts such as perspectives, views, editors, and builders as well as user interface elements such as menu-bars and tool buttons. The Platform has a large number of extensions points that make it possible to add functionality (e.g., with a new editor or item at the menu-bar). The JDT, building on the Platform, realizes a fully-featured Java IDE. The PDE extends the JDT with support for plug-in development.

Each layer consists of a number of plug-ins. The Eclipse Core defines the plug-in infrastructure and has functionality to discover and (lazily) load the required plug-ins. Besides this small kernel, all the other functionality is located in plug-ins. Gamma and Beck refer to this as the "everything is a contribution" rule [11], illustrated in Figure 4. Thus, it is possible to extend existing tools with new functionality as well as to build fully-featured tools. Eclipse 2.1 itself has about 100 plug-ins. The IBM WebSphere Application Developer (WSAD) 5.0 application adds more than 800 additional plug-ins.

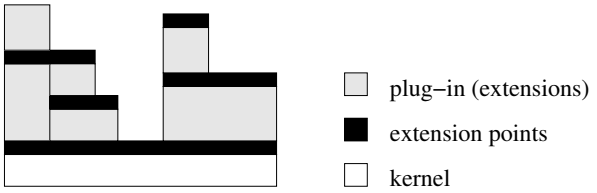


Figure 4. Eclipse Plug-In Architecture

Extension points in Eclipse are similar in spirit to extension mechanism offered in Montana (cf. Section 2.1)—one crucial difference being that the set of extensions in Montana is fixed by the baseline environment (i.e., single-level architecture), whereas any plug-ins in Eclipse can define extensions, leading to a tool architecture of many levels (i.e., multi-level architecture [19]).

In contrast to most of the components discussed previously, Eclipse is open source [29], which means that the full Java source code is available to any developer. A large number of developers contribute to the code base. One mayor contributor is Object Technology International, a subsidiary of IBM.

Various software engineering research tools are now based on Eclipse [10]. Examples are a CASE tool for the Business Object Notation [33], support for Test Driven Development [23], automated debugging [6], the porting of the ASF+SDF language development environment to Eclipse [37], and the Hipikat recommendation tool [7]. The Shrimp visualization tool [30]—primarily used for reverse engineering—has been integrated in Eclipse [18] and WebSphere [24].

3. Evaluation

This section discusses five characteristics of components that have a crucial impact on tool building:

1. steep learning curve (−)
2. missing customization functionality (−)
3. availability of customization examples (+)

4. component evolution (−)
5. enabling of rapid prototyping (+)

A "+" indicates that a certain characteristic is beneficial from a tool-builder's perspective; conversely, a "−" indicates an undesirable characteristic.

There are other characteristics for software—notably quality attributes such as maintainability and usability—that we do not address. Some of these attributes directly address the source code (e.g., maintainability), which is irrelevant for black-box components. Others address the end user (e.g., usability).

For a meaningful discussion, we chose to only discuss such characteristics where we could find published experiences from other researchers. We plan to further expand and refine the characteristics in future research.

3.1. Learning Curve

Using components can drastically reduce the development effort when building new (software engineering) tools. However, in order to customize a component productively a significant learning effort is often required. This is especially the case if the component offers sophisticated programmatic customization based on an API. For example, one designer of Montana relates the following experiences [15]:

> "Regardless of how good the programming interfaces are, [Montana] is a big system, and there is a steep learning curve. The programming interfaces are easy to learn, but we need to publish a book of idioms, a programmers guide. Once the idioms have been learned, tool writers can be very productive."

Part of the difficulty of learning the Montana API is explained by the complexity of the C++ language [21]. Even though the available extensions in Montana are a fixed set, it is a large number. The preprocessor, for example, has a relatively large number of about 50 observer extensions. Furthermore, there are three different extension mechanism (however, not all of these have to be necessarily understood).

In Eclipse, the number of extension points is even larger. In fact, since any plug-in can define extensions, the number of useful extensions for a tool-builder are open-ended. Not surprisingly, Eclipse developers have experiences that are similar to Montana's [6]:

> "The steep learning curve for plug-in developers in ECLIPSE is hard to master. . . . Once the learning curve is mastered, ECLIPSE turns into a very powerful platform."

Eclipse promises significant leverage for tool builders, but at the expense of a steep learning curve.

Lastly, it probably lessens the learning curve if a component can be explored via scripting, because the tool builder can more easily experiment and explore. If supported by the component, tool builders can also start with scripting and switch to statically compiled languages later on. Eclipse has plug-ins that support scripting functionality in JavaScript, however, the development has been suspended.

3.2. Customization Functionality

Customizations of components is limited by the functionality that the API and/or scripting language provides. Unfortunately, missing functionality is hard to identify upfront. Often, they materialize unexpectedly after substantial development efforts with a component has been already made. Various researchers have struggled with APIs that are not powerful enough for their needs. Reiss reports the following limitation in the FrameMaker API that he encountered while implementing the Desert environment (cf. Section 2.1):

> "While FrameMaker notified the API that a command was over, it did not provide any information about what the command did or what was changed" [25].

As a consequence, the user sometimes has to reload the editor contents (e.g., after a global find/replace). Reiss concludes that while FrameMaker's baseline environment is beneficial, it also "has restricted the interfaces we can offer" [26].

Similarly, Goldman and Balzer [13], who customized PowerPoint for visual editing, had to work around missing events:

> "Detecting events initiated through the native PowerPoint GUI was a serious problem. Although a COM interface *could* make relevant events available, the interface implemented by PowerPoint97 *does not*."

Despite their best efforts, they were not able to sufficiently capture the effect of some events (e.g., undo):

> "Although we may have control both before and after PowerPoint executes that action, we have no effective means, short of a complete comparison of before and after states, to determine the relevant state changes. The best we can do is simply remove such tools from the GUI."

Problems with components can show up in unexpected ways. When customizing Visio, we found that some predefined masters did pop up user dialogs [41]. The values

requested by the dialogs could not be given programmatically. In the end, we had to re-implement most of the master's functionality with low-level operations.

These problems are most severe for black-box components whose source code cannot be examined and modified. Eclipse gives tool-builders much more freedom. They can become part of the Eclipse community and influence the design of an API and/or inspect the current code and find a suitable workaround. If an official, documented API is not available, they can fall back to less stable internal APIs.

Interestingly, using components can expose deficiencies in one's own tool design. When porting the ASF+SDF Meta-Environment to Eclipse, the authors found that their editor interface could be streamlined [37].

3.3. Customization Examples

As discussed above, customizing a component often comes with a steep learning curve. The learning curve can be significantly flattened if supporting documentation that gives customization examples is available. Studies have shown that many users customize by adapting existing sample code (which is provided by experts in their organization) [12].

Code samples that show how an API is used most effectively can be made available with cookbooks, (online) articles, and discussion forums. Montana's developers propose a book of idioms to make tool builders more productive. Adobe maintains the FrameMaker Developer Knowledgebase, which has some typical examples for how to customize FrameMaker via its C API [1]. Despite such efforts, it can be surprisingly hard to find good sample code for a particular need. Especially for black-box components there is little production-strength code publicly available that one can use as a basis for study and adaptation; for instance, we made this experience when customizing Microsoft Office [40].

Following the observation that customization is often done by adaptation of existing code, Gamma and Beck propose to develop Eclipse plug-ins as follows [11, page 40]:

> "Monkey See/Monkey Do Rule: Always start by copying the structure of a similar plug-in."

Developers make indeed use of this strategy as illustrated by the following experience report of a tool builder [6]:

> "Often, if the programmer wants to accomplish something, she has to search the ECLIPSE source code for that functionality, copy it and customize it to suit her needs."

It is hard—or impossible—to apply the Monkey See/Monkey Do Rule for customizing black-box components. In contrast, since almost everything in Eclipse constitutes a plug-in there is a large amount of documented production-strength sample code available. Ironically, because of the huge amount of sample code, finding appropriate code can be overwhelming.

3.4. Component Evolution

If components and their APIs evolve, the customizations may have to be adapted as well, resulting in unexpected maintenance activities.

For tool builders it is desirable to have a predicatable evolution of the component and their APIs. Eclipse distinguishes between official and inofficial APIs. Extenders should exclusively use official APIs, which can only change for major releases. Internal APIs are clearly identified as such and their use is discouraged [9]; however, they represent a vital escape route not available for black-box components.

To give an example, the Eclipse plug-in that supports CVS access does not offer an official API. Xia is a research tool that provides visualizations of CVS information [18]. Xia wants to explore if visualizations can help collaborative efforts of programmers via user studies. In order to realize Xia, the tool builders report that

> "we access internal code from the CVS plugin. An API for the CVS plug-in is to be released soon, but there are absolutely no guarantees that our code will compile or behave similarly in the future releases of Eclipse or its included CVS plug-in. Internal code is of course used at one's own risk."

It is also desirable for tool builders to influence the evolution of the API by providing feedback. Tool builder cannot hope to influence the evolution of commercial black-box components that are controlled by one supplier (e.g., Microsoft Office or Lotus Notes). In contrast, the evolution of the observer extension in Montana were driven by tool building. Observer extension in Montana have been incrementally added and refined as requested by tool builders: "Instead of guessing all of the distinguished points for observation, we added observers as needed" [28].

While the feedback cycle in Montana was limited by the comparably small number of tools that have been written, the large, open, and active community of Eclipse has the potential to provide an effective feedback loop. For example, some researchers have pointed out limitation of Eclipse that they would like to see addressed [37] [19].

3.5. Rapid Prototyping

Using components for tool-building can enable rapid prototyping of tool functionality.

If the component is carefully selected, its baseline environment can cover a significant part of the tool functionality. This is the case, for example, with the Desert editor based on FrameMaker [26]:

> "FrameMaker provides many of the baseline features we needed: it displays both pictures and text; it can display high-quality program views; and many programmers are familiar with it, using it for documentation or specifications."

In such cases, significantly less code needs to be written and subsequently maintained.

The Rigi C++ extractor (cf. Section 2.1) was developed rapidly with Montana. Writing and maintaining a full C++ extractor from scratch is difficult and error-prone. In contrast, the Rigi extractor was written in less than two weeks and has under 1000 lines of C++. In contrast, the hand-coded Rigi C extractor has more than 4,000 lines of rather complex Lex, Yacc, and C++ code. The developer of the C++ extractor reports that

> "using the Montana CodeStore API helped to develop the parser much faster than if a parser had been written from scratch or by modifying a traditional compiler" [21].

We made similar experiences when implementing RENotes and REVisio (cf. Section 2.2). For RENotes, the JGraph component was used to implement the graph visualization [3]. This implementation has about 4,000 lines of Java code. In contrast, Rigi has about 30,000 lines of C/C++ code.

4. Conclusions

This paper explained the idea of tool-building with components and surveyed three typical approaches (API, scripting, and Eclipse). We discussed the strength and weaknesses of each approach according to five characteristics, drawing from researchers' published experiences.

While leveraging components can be an effective way to build research tools, tool builders have to be aware of the potential problems of this approach. More research in this area is needed, for example, lightweight development processes and strategies to effectively select among candidate components.

Most importantly, researchers should publish their tool-building experiences as part of their research papers. Unfortunately, many software engineering researchers do not report their experiences in papers because they do not consider them part of the publishable research results. We believe, however, that these experiences are valuable and could greatly benefit other researchers. This is especially the case in the software engineering arena, where tool-building is a fundamental research activity.

The characteristics introduced in the paper address the impact that component have from a tool-builder's perspective. We plan to further refine and extend these characteristics. Another important consideration, however, is the impact of components on the tool user. We believe that both parts of the coin are equally important. Future research will try to identify desirable characteristics from the users' perspective. Examples of such desirable characteristics are tool interoperability [42] and cognitive support [39]. Furthermore, characteristics identified by diffusion of innovations theory (e.g., relative advantage, compatibility, and trialability) are a promising starting point for further investigations [27].

Acknowledgments

Thanks to Crina-Alexandra Vasiliu for proofreading. This work has been supported by the IBM Toronto Center for Advanced Studies (CAS), the Natural Sciences and Engineering Research Council of Canada (NSERC), and the Consortium for Software Engineering (CSER).

References

[1] Adobe. Framemaker developer knowledgebase. `http://support.adobe.com/devsup/devsup.nsf/framekb.htm`.

[2] Adobe. Framemaker developer's kit overview. `http://partners.adobe.com/asn/framemaker/fdk.jsp`.

[3] G. Alder. JGraph home page. `http://jgraph.sourceforge.net/`.

[4] Projekt Bauhaus. `http://www.informatik.uni-stuttgart.de/ifi/ps/bauhaus`.

[5] R. Biddle, A. Martin, and J. Noble. No name: Just notes on software reuse. *ACM SIGPLAN Notices*, 38(2):76–96, Feb. 2004.

[6] P. Bouillon, M. Burger, and A. Zeller. Automated debugging in Eclipse (at the touch of not even a button). *2003 OOPSLA workshop on eclipse technology eXchange*, pages 1–5, Oct. 2003.

[7] D. Cubranic and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. *25th International Conference on Software Engineering (ICSE 2003)*, pages 408–418, May 2003.

[8] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[9] J. des Rivieres. How to use the Eclipse API. *Eclipse Corner*, May 2001. `http://www.eclipse.org/articles/Article-API%20use/eclipse-api-usage-rul%es.html`.

[10] Eclipse. Eclipse research community. `http://www.eclipse.org/technology/research.html`.

[11] E. Gamma and K. Beck. *Contributing to eclipse*. Addison-Wesley, 2004.

[12] M. Gantt and B. A. Nardi. Gardeners and gurus: Patterns of cooperation among CAD users. *Conference on Human Factors in Computing Systems (CHI 92)*, pages 107–117, May 1992.

[13] N. M. Goldman and R. M. Balzer. The ISI visual design editor generator. *IEEE Symposium on Visual Languages (VL '99)*, pages 20–27, Sept. 1999.

[14] W. Gregg. Hypercard: Hypercard extends the Macintosh user interface and makes everybody a programmer. *Byte*, pages 109–117, Dec. 1987.

[15] M. Karasick. The architecture of Montana: An open and extensible programming environment with an incremental C++ compiler. *6th ACM SIGSOFT international symposium on Foundations of software engineering (FSE-6)*, pages 131–142, Nov. 1998.

[16] R. Kazman and S. J. Carriere. Playing detective: Reconstructing software architecture from available evidence. *Journal of Automated Software Engineering*, 6(2):107–138, Apr. 1999.

[17] H. M. Kienle, A. Weber, J. Martin, and H. A. Müller. Development and maintenance of a web site for a bachelor program. *5th International Workshop on Web Site Evolution (WSE 2003)*, Sept. 2003.

[18] R. Lintern, J. Michaud, M.-A. Storey, and X. Wu. Plugging-in visualization: experiences integrating a visualization tool with Eclipse. *2003 ACM Symposium on Software visualization (SoftVis '03)*, pages 47–56, June 2003.

[19] C. Lüer. Evaluating the Eclipse platform as a composition environment. *3rd International Workshop on Adoption-Centric Software Engineering (ACSE 2003)*, pages 59–61, May 2003.

[20] J. Ma, H. M. Kienle, P. Kaminski, A. Weber, and M. Litoiu. Customizing Lotus Notes to build software engineering tools. *CASCON 2003*, Oct. 2003.

[21] J. Martin. Leveraging IBM VisualAge for C++ for reverse engineering tasks. *CASCON '99*, pages 83–95, Nov. 1999.

[22] J. Martin and L. Martin. Web site maintenance with software-engineering tools. *3rd International Workshop on Web Site Evolution (WSE 2001)*, pages 126–131, Nov. 2001.

[23] N. Nagappan, L. Williams, and M. Vouk. "Good enough" software reliability estimation plug-in for Eclipse. *2003 OOPSLA workshop on eclipse technology eXchange*, pages 30–34, Oct. 2003.

[24] D. Rayside, M. Litoiu, M.-A. Storey, and C. Best. Integrating SHriMP with the IBM WebSphere Studio workbench. *CASCON 2001*, pages 79–93, Nov. 2001.

[25] S. P. Reiss. Program editing in a software development environment (draft). 1995.

[26] S. P. Reiss. The Desert environment. *ACM Transactions on Software Engineering and Methology*, 8(4):297–342, Oct. 1999.

[27] E. M. Rogers. *Diffusion of Innovations*. The Free Press, fourth edition, 1995.

[28] D. Soroker, M. Karasick, J. Barton, and D. Streeter. Extension mechanisms in Montana. *8th Israeli Conference on Computer Systems and Software Engineering (CSSE '97)*, pages 119–128, June 1997.

[29] D. Spinellis and C. Szyperski. How is open source affecting software development? *IEEE Software*, 21(1):28–33, Jan./Feb. 2004.

[30] M.-A. Storey, C. Best, and J. Michaud. SHriMP views: An interactive environment for exploring java programs. *9th International Workshop on Program Comprehension (IWPC 2001)*, pages 111–112, May 2001.

[31] T. Systä, K. Koskimies, and H. Müller. Shimba—an environment for reverse engineering Java software systems. *Software—Practice and Experience*, 31(4):371–394, 2001.

[32] T. Systä, P. Yu, and H. Müller. Analyzing Java software by combining metrics and program visualization. *4th European Conference on Software Maintenance and Reengineering (CSMR 2000)*, Feb. 2000.

[33] A. Taleghani and J. Ostroff. BON development tool. *2003 OOPSLA workshop on eclipse technology eXchange*, pages 10–14, Oct. 2003.

[34] S. R. Tilley. Domain-retargetable reverse engineering II: Personalized user interfaces. *1994 International Conference on Software Maintenance (ICSM '94)*, pages 336–342, Sept. 1994.

[35] S. R. Tilley, H. A. Müller, M. J. Whitney, and K. Wong. Domain-retargetable reverse engineering. *Conference on Software Maintenance (CSM '93)*, pages 142–151, Sept. 1993.

[36] T. Tulisalo, R. Carlsen, A. Guirard, P. Hartikainen, G. McCarthy, and G. Pecly. *Domino Designer 6: A Developer's Handbook*. IBM Redbooks, Dec. 2002.

[37] M. G. J. van den Brand, H. A. de Jong, P. Klint, and A. T. Kooiker. A language development environment for Eclipse. *2003 OOPSLA workshop on eclipse technology eXchange*, pages 55–59, Oct. 2003.

[38] E. van Emden and L. Moonen. Java quality assurance by detecting code smells. *9th Working Conference on Reverse Engineering (WCRE 2002)*, Oct. 2002.

[39] A. Walenstein. Improving adoptability by preserving, leveraging, and adding cognitive support to existing tools and environments. *3rd International Workshop on Adoption-Centric Software Engineering (ACSE 2003)*, pages 36–41, May 2003.

[40] A. Weber, H. M. Kienle, and H. A. Müller. Live documents with contextual, data-driven information components. *SIGDOC 2002*, Oct. 2002.

[41] Q. Zhu, Y. Chen, P. Kaminski, A. Weber, H. Kienle, and H. A. Müller. Leveraging Visio for adoption-centric reverse engineering tools. *10th Working Conference on Reverse Engineering (WCRE 2003)*, Nov. 2003.

[42] D. M. Zwiers. Improving interoperability to facilitate reverse engineering tool adoption. Master's thesis, Department of Computer Science, University of Victoria, Apr. 2004.

On the Challenges in Fostering Adoption via Empirical Studies

Shihong Huang
Department of Computer Science
University of California, Riverside
shihong@cs.ucr.edu

Scott Tilley
Department of Computer Sciences
Florida Institute of Technology
stilley@cs.fit.edu

Abstract

This paper outlines some of the issues in fostering adoption via empirical studies. There has been a welcomed shift in recent years towards the use of evidence-based techniques to support arguments of efficacy of proposed software engineering tools and techniques. When available, such objective indicators can provide a strong argument for (or against) the adoption of a particular approach. Unfortunately, gathering such evidence is not a trivial matter. Several of the key issues encountered in conducting a series of limited-scope qualitative experiments are used to illustrate the hurdles that must be overcome before empirical studies can enjoy more widespread use.

Keywords: *adoption, empirical studies, evidence-based software engineering, education*

1. Introduction

One of the central tenets of software engineering is to adopt a more disciplined approach to the production of software artifacts [4]. There are many stated benefits to such an approach, such as increased quality, improved effort estimation, and lower maintenance costs. However, even with such impressive benefits, getting practitioners to adopt the methods is problematical without indisputable proof that the cost of adoption is outweighed by the return on their investment.

In other words, there is a need to employ evidence-based arguments to support the adoption of best practices in software engineering, rather than arguments based upon advocacy [8]. Objective evidence would facilitate adoption by providing an independent predictor of the likely benefits of the proposed approach. Unfortunately, many researchers do not provide such objective evidence. The result is that there is rarely any solid audit trail that can provide a validation of the ideas, and which links theory and concepts to observed practices [1].

One reason for this lamentable situation may be a lack of understanding on the part of many researchers in the academic community as to how to produce such evidence. For example, recent work in the reengineering of transactions for Web-based systems shows promise [2], but without a comprehensive case study the research is not as convincing as it might otherwise be. One of the most powerful techniques that could be used in this situation is an empirical study, but successfully realizing such a study is fraught with difficulties.

Our own personal experience in conducting two limited-scope qualitative empirical studies assessing the efficacy of UML as one type of graphical documentation in aiding program understanding supports this observation [6][7]. We are novice empiricists who are interested in using empirical studies as a measurement instrument; we are not particularly interested in becoming experts concerning empirical methods per se. As novices, we struggled with learning how to properly execute an empirical study, while at the same time remaining focused on the underlying problem that we were interested in exploring.

Of the numerous challenges in fostering adoption by means of empirical studies, one of the most vexing is dealing with threats to validity. All empirical studies can have possible threats to the validity of the experiment and the conclusions drawn from the analysis of the results. For qualitative experiments, these threats can be managed somewhat more easily than with quantitative experiments. Nevertheless, proactively addressing any possible doubts that others might have concerning the credibility of reported results is essential for using the experiemnt as a means of fostering adoption of the tool or technique that was the focus of the study.

The next three sections of the paper discuss specific challenges related to dealing with threats to validity for empirical studies: designing the experiment, selecting the participants, and choosing the sample system. These challenges are described from a personal point of view, and are certainly not meant to be exhaustive. The paper concludes by outlining possible avenues for future work to addresses these challenges.

2. Designing the Experiment

The first threat to validity is the nature of the experiment itself. The threat begins with questioning the basic premise for conducting the experiment: the hypothesis. In essence, this threat is concerned with verifying that the right question is being asked. It is fruitless to expend the energy needed for an empirical study on a question that has a patently obvious answer. At the same time, knowing which question to ask is not easy. There is a constant tug between asking a very specific question (whose answer may be accurate but not broadly applicable), and asking a more general question (whose answer may be subjective and hence would not hold up to third-party scrutiny).

In other disciplines, experiments are often tightly controlled, with a single variable the subject of investigation; all other variables are held constant. This approach is also possible in a software engineering experiment, but it may not be the most desirable choice. An experiment that is so narrowly focused may indeed produce objective results using a repeatable procedure, but those results may not support the broader goal of adoption. In contrast, an experiment that more closely recreates the conditions of an actual development project will necessarily have many variables, only one of which may be of interest to the researchers. Any results arising from the analysis of such an experiment can be questioned due to the uncertain traceability between cause and effect.

3. Selecting the Participants

The second threat to validity is the selection of participants in the experiment. A hard reality for most academic researchers is that the people available to participate in experiments are either students or colleagues. The advantage of having such participants is that one might be more knowledgeable about the skills and experience of each person, which could be valuable information when it comes to evaluating personal biases in the results.

The disadvantage of using friends and/or students in empirical studies is that such people are not necessarily representative of the society at large. If the goal of the experiment is to provide evidence to support widespread adoption of a particular tool or technique, it is important to have participants who reflect the likely characteristics of the larger body of users. Otherwise, arguments for (or against) efficacy will be tainted by the undue influence of the participants' skills (or lack thereof).

There is also the issue of scale: results from an experiment with only a handful of participants can rightly be called into question. In other fields, such as medicine, empirical studies often involve hundreds or thousands of participants. In most software engineering studies, researchers are often pleased if they can attract a dozen students to participate in their experiment. The statistical significance of such a small number of participants is questionable. But sometimes the results are still of value.

For small-scale experiments, the use of students as the primary experimental participants may be acceptable. However, in the context of fostering adoption, it is more desirable to involve people from the other two pillars of the community: government and industry. Unfortunately, it is notoriously difficult to get representatives from these two groups to participate in academic experiments. The reasons for this are many and varied, from lack of interest to lack of time. One technique that has proven successful is to change the image of the experiment from an exercise purely of interest to the academic, to a collaborative activity between equal partners. Carnegie Mellon's Software Engineering Institute adopted this approach during the development of the Software Capability Maturity Model [5] with great success.

4. Choosing the Sample System

The third threat to validity is choosing the sample system. Most academic empirical studies in software engineering are of limited scope (that is, performed during a short time frame with a relatively small number of participants). This limitation usually forces the researchers to make a choice between a sample system that is representative of a real-world problem, and a much smaller application that is (hopefully) still representative but much simpler.

The problem with choosing a sample system that is as close as possible to a real-world application is that the complexity inherent in such systems may be too much for the participants to master within the context of the experiment. In contrast, choosing a smaller and simpler sample system may make it easier for the participants to work with the material, but at the cost of potentially exposing the results to prejudice based on "toy" experiments – easy to run, but with results that are difficult to generalize.

The choice of the sample system is also complicated by the nature of the experiment's design. It is quite difficult to control "variables" in a scientific sense for most software engineering experiments. For example, if one is interested in evaluating the usefulness of a particular form of graphical documentation in support of program understand, the sample system should be complex enough that a typical software engineer would have some difficulty in understanding the application without the aid of such documentation. However, there is no such thing as a "typical" software engineer; previous

experience, knowledge of the application and implementation domain, and personal preferences for cognitive support can all influence analysis of the results.

The ability to reproduce the results of the same experiment by a different experimenter is one of the trademarks of a scientific study. One way to encourage this in software engineering may be to use an open source project as the sample system. This would seem to offer a commendable level of visibility into the experiment. However, using a commercial application also has advantages, particularly in terms of adoption: industry may be more receptive to results that derive from analysis of one of their own applications, as opposed to one that lacks the very important "ownership" attribute. Ownership leads to buy-in, and buy-in leads to adoption.

5. Summary

This paper briefly outlined three challenges related to threats to validity of empirical studies: designing the experiment, selecting the participants, and choosing the sample system. These challenges were identified from our own experience in organizing limited-scope empirical studies in software engineering.

There are of course many other challenges related to empirical studies outside of those concerning threats to validity, particularly in using the results of the experiments to foster adoption. For example, addressing ethical issues pertaining to treatment of the participants, running the experiment itself, and analyzing the results. Addressing these challenges requires education of both the researchers and the consumers of the results.

During the preparation for our own empirical studies we felt the need for a source of guidance on performing such studies, perhaps in the form of a handbook. There is a clear need for pedagogical material specifically targeted towards empirical studies in software engineering. The tutorial "Case Studies for Software Engineers" that is part of the ICSE 2004 conference program [3] is an excellent step towards satisfying this goal.

References

[1] Budgen, D.; Hoffnagle, G.; Müller, M.; Robert, F.; Sellami, A.; and Tilley, S. "Empirical Software Engineering: A Roadmap." *Proceedings of the 10th International Conference on Software Technology and Engineering Practice* (STEP 2002: Oct. 6-8, 2002; Montréal, Canada), pp. 180-184. Los Alamitos, CA: IEEE Computer Society Press, 2003.

[2] Distante, D.; Parveen, T.; and Tilley, S. "Towards a Technique for Reverse Engineering Web Transactions from a User's Perspective." *Proceedings of the 12th IEEE International Workshop on Program Comprehension* (IWPC 2004: June 24-26, 2004; Bari, Italy). Los Alamitos, CA: IEEE CS Press, June 2004.

[3] Perry, D.; Sim, S; and Easterbrook, S. "Case Studies for Software Engineers." Tutorial presented at *The 26th International Conference on Software Engineering* (ICSE 2004: May 23-28, 2004; Edinburgh, Scotland, UK).

[4] Pfleeger, S. *Software Engineering Theory and Practice* (2nd Edition). Upper Saddle River, NJ: Prentice-Hall, 2001.

[5] Software Engineering Institute, Carnegie Mellon University. "CMMI Adoption." Online at http://www.sei.cmu.edu/cmmi/adoption/.

[6] Tilley, S. and Huang, S. "Assessing the Efficacy of Software Architecture Visualization Techniques for Recovered Artifacts." *Dagstuhl Seminar 03061: Software Architecture Recovery and Modeling* (Feb. 2 – 7, 2003; Schlöss Dagstuhl, Germany).

[7] Tilley, S. and Huang, S. "Workshop on Graphical Documentation for Programmers: Assessing the Efficacy of UML Diagrams for Program Understanding." Held in conjunction with *The 11th International Workshop on Program Comprehension* (IWPC 2003: May 10, 2003; Portland, OR).

[8] Tilley, S.**;** Huang, S.; and Payne, T. "On the Challenges of Adopting ROTS Software." *Proceedings of the 3rd International Workshop on Adoption-Centric Software Engineering* (ACSE 2003: May 9, 2003; Portland, OR), pp. 3-6. Published as CMU/SEI-2003-SR-004. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, June 2003.

Value Assessment by Potential Tool Adopters: Towards a Model that Considers Costs, Benefits and Risks of Adoption

Timothy C. Lethbridge
School of Information Technology and Engineering, University of Ottawa, Canada
tcl@site.uottawa.ca

Abstract

When a user starts to think about whether she or he ought to adopt a new software engineering tool, she or he will think about costs, benefits and risks. The thinking may not be deep and rigorous, but it is clear from prior research that a certain amount of such thinking takes place. In this paper we present a model that categorizes the costs, benefits and risks and shows how they relate in a simple inequality. This can be used to guide tool developers towards creating more adoptable tools, and might be an improvement on current approaches that tend to address only a limited number of issues at a time. We show how this model can be applied towards adoption scenarios such as adoption of a commercial tool and adoption of Eclipse.

1. Introduction

One of the beliefs that motivates software engineering tool builders is, "if we build it, they will come." Unfortunately, they often don't come, and we wonder way. Some blame usability, or the difficulty of making a tool usable due to difficulties with underlying technology; some believe we cannot force users to change their habits; and some think that users just don't know what is best for them.

All of these things may be true to some extent, some of the time. However, this paper proposes that a more comprehensive understanding of adoptability can be obtained by thinking more broadly about what we call the *value proposition*: The set of conclusions a potential adopter must reach that lead them to think that it would be worthwhile to adopt a tool in a given context.

In this paper, we first examine a few of the recent arguments that have been made for why tools are not adopted, and a few of the proposals for addressing these problems. Next we present our value proposition model. Finally we show how it can be used to understand specific cases of adoptability and adoption (or the lack thereof).

2. Solutions proposed for adoption problems, and a word of caution

Tilley et al [1] suggest that tools, especially those from academic environments might be more adoptable if they were more understandable, robust, and complete. Martin [2] suggests a tool ought to be integrated, be responsive and be flexible. Favre et al [3] raise additional concerns such as scalability.

The above concerns relate to *technical* improvements that can be made to tools. Others discuss the *social* issues; for example, Roth [4] suggests that humans naturally tend to resist change. This is also, undoubtedly, part of the answer. Favre et al [3] raise concerns about administration, evolution, training and vendor-dependence as obstacles to adoption.

Various comprehensive solutions have been proposed. One that is at the forefront of current thinking, and that in fact is a major theme of this workshop, is that one can improve tool quality and overcome the resistance to change by building software development tools inside widely used off-the-shelf products such as office programs (word processors, spreadsheets, etc.). Kienle [5], for example, justifies this by pointing to the fact that tool developers struggle to develop the "bread and butter" data manipulation and UI capabilities that are already well developed in office products and are well understood by most users.

We are, however, not convinced that this will be a panacea since many users of office software do not adopt important features of that software. For example, we have noticed many users of Microsoft Word not using styles. The intended use of styles is that almost every piece of text is assigned one of several document styles so the document can be restyled easily. However, many ordinary Word users do not use styles at all, or accidentally make the limited use of styles imposed by

the 'automatically apply styles' feature. Other users make use of styles in a piecemeal manner – perhaps just for top level headings. Piecemeal users will happily, however, directly apply font and emphasis modifications to modify the look of large parts of their document without updating the styles in a disciplined manner.

Similarly we have observed that relatively few Microsoft Excel users take full advantage of its many data-organizing capabilities.

So if users of office products often don't adopt a product's own features, why should we expect them intrinsically to be more likely to adopt a software engineering tool just because it is built in an office product.

We do believe that all of the solutions discussed in this section have merit. However, we also believe that their proponents are sometimes too quick to believe that a given solution will solve the adoption problem.

There is a long history of general research into adoption or non-adoption of technological innovations. Perhaps the most important book in the field is *Diffusion of Innovation*, by Rogers [6]. In that book, Rogers points out that there are numerous reasons why individuals do not adopt innovations, or adopt them slowly. One of his key points is that an individual's adoption decision is based on his or her *perception* of various factors, rather than absolute truths about those factors. We will, in several places throughout this paper, show how some points made by Rogers can be applied in the context of software engineering tools.

In Rogers' model, adoption starts with the *knowledge stage*, in which people become aware of the existence of an innovation. Then they move to the *persuasion stage* in which they form 'favorable or unfavorable attitude' towards the innovation, perhaps influenced by marketing or interaction with others. Only after forming a favorable attitude do they move to the *decision stage*, where they consciously or subconsciously consider various factors that may lead to trial use or more intense use. In this paper we will focus on the issues considered during the decision stage.

3. Costs, benefits and risks associated with usage and adoption of a tool

In this section, we will start to develop a model that accounts for some of the decisions a potential adopter must make. We will relates these in such a way that a tool builder can perhaps better assess what to do to improve adoption.

Note that although we refer to the 'tools' in what follows, much of this discussion can apply to any software, not just software engineering tools. We also don't intend what follows to be considered a rigorous mathematical model, rather it is a general model to aid in thought. We are also proposing it as a first iteration, with the hope that it will form the basis for discussion and extension.

We will model tool use and adoption using the variables Cu (*cost of use*), Bu (*benefit of use*) and Ru (*risk of use*). Below are some of the factors that can be summed to determine the values of these variables.

Cost of use, **Cu**, can have many components, including the following:

- Costs that are primarily paid *once* when use of the tool starts; we call these costs of adoption, Ca:
  1. Up-front financial cost for the software (if bought), or initial development cost of the software (if developed in-house). We believe that this factor is one of the biggest obstacles to adoption of many commercial software engineering tools, since people will only be willing to pay for software when they are convinced they will make good use of most of it. Rogers [6] points out that innovations are more adoptable if they can be divided (i.e. partially adopted, or adopted to the extent that users need them).
  2. Financial cost of any extra hardware or support software required to use the tool. This factor sometimes lurks in the background because new tools can tax the power of the current generation of computers.
  3. Time to install and configure the tool.
  4. Time and financial cost to convert any data, if required, before using the tool.
  5. Time for each user to learn how to use the tool.
- Costs that must be paid on an ongoing basis to use the tool
  6. Incremental extra time required to use the tool as opposed to doing the task using existing approaches. We will only consider this variable to have a positive or zero value – if the value would be negative then it is part of the benefit of use. One might expect that for a tool to be at all useful, this cost ought to be zero. However, there can be situations when benefits accruing from a tool can justify adoption of a tool that actually takes longer to use than the status quo.

7. Ongoing time and financial cost of being 'different' from others (e.g. ongoing data conversion costs).
8. Ongoing time and financial cost to maintain the tool (e.g. updating versions, maintaining a database)

Benefit of use, **Bu**, can also have many components, including the following.

1. Incremental time saved by doing a task using the tool. This value greater than zero if the tool will enable the user to work faster. Improving usability, robustness and responsiveness will help here.
2. Time saved in the long run. For example, having set up one's design using a certain tool, one can then perhaps find information faster later.
3. Value of the increased quality of work done.

Finally, the **risk of use, R u**. has many possible components

1. Risk that any of the components of Cu will be greater than expected
2. Risk that any of the components of Bu will be less than expected
3. Risk that use of the tool will have unexpected negative side-effects (e.g. corrupting data)
4. Risk that it will be difficult to revert to previous tools if tool under consideration is not a success (e.g. because the user has started to store data in a certain format)
5. Risk that support for the tool will be dropped or inadequate, or that tool upgrades will result in obsolete data.
6. Risk of encountering defects, including usability defects, poor documentation, crashes and 'bugs'.
7. Risk that the tool will not allow the user to do the task in the way they want
8. Risk that the adopter will not be able to tell whether or not the tool made any difference.

Note that many of the costs, benefits and risks are not only dependent on a particular tool, but also on a particular context of use. For example benefits 2 and 3 in general will only appear if the user actually does a good job.

Rogers model of adoption [6], discussed in the last section, is a more general model of which the above is a special case. Rogers identifies five main factors that the potential adopter needs to perceive when making an adoption decision:

Compatibility: An adopter will be less likely to adopt an innovation that he or she perceives will requires major changes to the way he or she does things. This is reflected in Cu4, Cu5 and Cu7 as well as Ru4 and Ru7.

Complexity: Innovations perceived to be simpler are more adoptable: This is reflected in Cu5 and Cu8.

Trialability: Adoption is more likely if the adopter perceives they can use the innovation a little and stop if the trial proves unsatisfactory. This is reflected in Cu1, Cu3, Cu4 and Cu5 as well as Ru1, Ru3 and especially Ru4.

Observability: An adopter needs to have the sense that they will be able to tell whether the innovation is benefiting them or not. This is reflected in Ru8.

Relative advantage: This is the topic of section 4, below, in which we will discuss how Cu, Bu and Ru can be related.

4. The value proposition: Inequalities to model use and adoption

Both the potential tool developer and the tool adopter must be convinced that there is value in the tool. Otherwise they won't bother developing it nor adopting it, respectively. The adoption problem stems from the fact that while the developer is convinced the tool has value, end users often are not.

In our model, we consider that both parties construct a value proposition to convince themselves of the usefulness of the tool, or the lack thereof. However, each party's value proposition will be a little different. We will examine, in general terms, two inequalities that can be used to model the value propositions.

The first inequality applies to the person making a decision to adopt and use a tool, who may be an individual user or a team leader. In order to choose to adopt a tool, such people must come to believe (very roughly) that:

$$pBu > (pCu * (1 + pROI)). \qquad \text{(Eq. 1)}$$

Where pBu is the *perceived benefit of use* of the tool, pCu is the *perceived cost of use* of the tool, and pROI is the *perceived return on investment*. As with any investment, pROI has to be greater than zero to account for perceived risks of use, pRu. The variables pBu, pCu and pRu are based on the variables Bu, Cu and Ru discussed in the last section, except that instead of being based on hard facts, they are based on perceptions.

As does Rogers [6], we use the term 'perception' since it is clear that users rarely make dry mathematical calculations about the costs, benefits and risks of

adopting a tool. Instead they normally just have a 'gut feeling' about these matters.

Those engineering a new tool ought therefore to consider the costs, benefits and risks that will be perceived by users. Improvements to adoptability can be achieved by not only improving the actual benefits and costs, but also by helping to ensure that perception more closely relates to reality.

Tool engineers may be convinced, with good reason, that it would benefit software engineers to adopt a given tool. However, we must recognize that potential adopters will perceive costs and benefits differently from what the engineers perceive those costs and benefits to be. Rogers [6] uses the term *heterophily* to describe the differences in background and thought processes of adopters versus innovators. In addition to perceiving costs and benefits differently. adopters will more intensively perceive the risks, and the more risks they perceive, the more their perceived benefits must exceed their perceived costs for adoption to take place. Mitigating the risks should therefore be given high priority, as should helping the potential adopter quantify the costs, benefits and risks.

While the perceived variables above are what actually drive an adoption decision, and will help engineers to improve adoptability. Scientists in the academic community studying tools often care more about whether a tool would be truly useful (or not) in a more pure or theoretical sense. They tend to care simply that:

$$Bu > Cu \qquad \text{(Eq. 2)}$$

Although in a scientific analysis of pure usefulness we know that there will be risks, the risks to the adopter that require the return on investment in Eq.1, can be factored out. For example, a scientific investigator will hopefully attempt to assess the actual costs and benefits by empirical studies (so we will not consider Ru factors 1 and 2). Such a person will also consider the theoretical case of a tool with sufficient quality such that there is no need not consider the remaining risk factors. Equation 2 can therefore be used to justify doing research and development, but it will not be sufficient to ensure that adoption takes place. Those in the academic community who also develop tools need to be aware of this.

We hypothesize that we can improve adoptability and adoption by better understanding the model presented in this section and the last. In the next section we will apply the model to various adoption scenarios.

5. Applying the model to specific scenarios

The model proposed in this paper can be applied to various scenarios. The first three scenarios below are those that we have observed in our research. The fourth is a scenario popular in the software engineering tool adoption literature.

5.1 Adoption by 'new hires' versus 'old hands'

We have observed repeatedly that the best time to achieve adoption of a tool is when people first start work. We were able to achieve adoption of our TkSee tool [7] by several new hires at Mitel, but there was little adoption by those we will call 'old hands'. People new to a project have to adopt *some* tool to do a given task, therefore Cu-1 to Cu-5 will be paid for any tool. Old hands however will have 'sunk costs' for the adoption of their current tool, and so will be reluctant to spend more on a new tool; unfortunately it is not in human nature to walk away from investments, even though it might be best to do so.

5.2 Users considering adopting Rational Rose

We have observed several people wishing to develop software by starting with various UML modeling tools. In one case, for example, a decision was made against adopting Rational Rose. Dominant costs were Cu-1 (they were paying a lot for a tool, of which they would be using only a fraction), and Cu-7 (exporting the data to other tools would be time consuming). Dominant risks were Ru-4 (conversion to other UML format tools appeared difficult), Ru-6 (the initial experience with usability was low, so the users feared continuing), and Ru-7 (the tool did not appear to allow certain free form editing the users desired).

5.3 Adoption of open integrated tool frameworks such as Eclipse

The Eclipse framework [8] has tremendous momentum as the preferred tool platform of many developers. We have noticed that developers are often very quick to adopt it, despite the fact that they will be having to make major changes to their existing process. They must therefore be able to make a clear value proposition.

We hypothesize that this could be due to the following; Cu-1 is zero (the tool is free), Cu-3 is very low (the tool runs out of the box), Cu-5 seems low compared to many other tools (great care has been spent making Eclipse usable). Cu-7 is low since the

tool works with ordinary files and a lot of people are starting to use the tool.

On the benefit side, the integration of a great many facilities results in a perception that lots of time will be saved. Most risks appear to be low as well; the open-source nature of the tool mitigate against Ru-5 (lack of support) or Ru-6 (encountering defects). Finally, Ru-7 (lack of flexibility) should be low due to the programmability of the tool and the large number of plugins becoming available.

We noticed a similar adoptability profile many years ago in the context of the Smalltalk programming environment (which was also open source, although proprietary).

5.4 Building tools in office products

As discussed at the start of this paper, an important idea in the tool adoption community is leveraging the cognitive support of office tools by developing software engineering tools inside them.

Doing this will clearly result in Cu-1 and Cu-2 being relatively low – most users already have office tools. However the approach does not *necessarily* lower some other costs: For example Cu-3 (installing the SE tool within the office product) is not necessarily going to be easy, nor is Cu-4 (data conversion). The big hope is that Cu-5 (learning time) will be low and Bu-1 (incremental time saved) will be high. Potential adopters will also likely consider Ru-4 (becoming trapped in a certain data format), and Ru-5 (that updates to office products will render the plugin tool inoperative).

Those developing SE tools inside office products therefore ought to consider all these aspects of the user's value proposition, and any others that might be applicable.

6. Conclusion

In this paper we have proposed that tool developers should consider all aspects of a user's perceived value proposition when analyzing whether a tool is likely to be adopted, or when understanding why it is not being adopted.

We presented a model of the factors users may consider when making their value proposition. We do not claim this model is definitive; however it is as a first step. We believe that much additional empirical research is needed; many examples of such studies can be found in Rogers [6].

Some of the things that our model can lead tool developers to consider are:

- Potential adopters *perceptions* of costs, benefits and risks are what affect their decision making. Understanding and influencing these perceptions is therefore important.
- Since potential adopters are somewhat risk averse, the benefit of a tool must considerably outweigh its cost. Adoption can therefore be improved by reducing the real risks, reducing the perception of risk, increasing the perceived benefit high enough or reducing the perceived cost low enough.
- There are many categories of costs, benefits and risks, all of which should be considered.

Acknowledgements

We would like to thank the anonymous reviewer who suggested integrating the ideas of Rogers [6] into the paper.

References

[1] Tilley, S., Huang, S. and Payne, T. (2003) "On the Challenges of Adopting ROTS Software", *3rd International Workshop on Adoption-Centric Software Engineering*, CMU/SEI-2003-SR-004.

[2] Martin, J. (2003) "Tool Adoption: A Software Developer's Perspective", *3rd International Workshop on Adoption-Centric Software Engineering*, CMU/SEI-2003-SR-004.

[3] Favre, J.-M., Estublier, J., Sanlaville, R. (2003) "Tool Adoption Issues in a Very Large Software Company", *3rd International Workshop on Adoption-Centric Software Engineering*, CMU/SEI-2003-SR-004.

[4] Roth, G., and Kleiner, A. (2000). *Car launch: The human side of managing change.* Oxford University Press, New York, NY.

[5] Kienle, H.M., and Janke, H. (2003) "A Visual Language in Visio: First Experiences", *3rd International Workshop on Adoption-Centric Software Engineering*, CMU/SEI-2003-SR-004.

[6] Rogers, E.T. (2003) *Diffusion of Innovation*, Fifth Edition, Free Press.

[7] Lethbridge, T.C. (2000) "Integrated Personal Work Management in the TkSee Software Exploration Tool", *Second International Symposium on Constructing Software Engineering Tools (CoSET2000)*, in association with ICSE 2000, Limerick, Ireland, pp 31-38.

[8] Eclipse, www.eclipse.org

On Understanding Software Tool Adoption Using Perceptual Theories

Dabo Sun and Kenny Wong
Department of Computing Science
University of Alberta, Canada
{dabo,kenw}@cs.ualberta.ca

Abstract

Successful adoption depends on many factors, one of which is cognitive support. This paper presents some theories in cognitive science, especially the theory of perception that might be useful to understand tool adoption. We study SHriMP, a software visualization environment and propose some suggestions to improve the visualization in SHriMP based on perceptual theories. We believe that combining research in cognitive science and software engineering will help us to evaluate existing tools and provide design criteria so that adoption will be improved.

1. Introduction

The main objective of software engineering research is to solve real-world problems and improve productivity. However, researchers have observed that most of the tools developed in academia often fail to be adopted by industry [11]. One of the reasons why research tools are not adopted is that they lack cognitive support [20, 22]. Although why people adopt tools is still not well-known, we believe that the combination of research in cognitive science and computing science will help us understand and improve tool adoption.

Many researchers have noticed the importance of cognitive support in software tools: Storey et al. did a user study trying to discover what kind of strategies programmers use to understand large software systems and how program comprehension tools help programmers understand programs [18]. They also generalized several cognitive models of program comprehension and suggested cognitive design elements to improve program comprehension and to reduce cognitive overhead [17]. Walenstein proposed a distributed cognition framework which treats a human and a computer as a distributed computational system and analyzed various cognitive problems in different domains [19]. Petre and Blackwell did a user study on ten expert programmers and studied the mental imagery they used to solve difficult problems [13]. Petre and Blackwell also discussed a variety of cognitive questions in software visualization to provide suggestions to tool designers [12]. Zayour and Lethbridge state the important role of cognitive aspects and psychological elements in designing successful reverse engineering tools. They presented a software tool called DynaSee to trace cognitive difficulties during software maintenance [22]. Ware surveyed the literature in human perception and provided design principles to visualize information effectively [21].

This paper focuses on cognitive theories of human perception and discusses how perceptual principles can help us design and evaluate tools. The rest of this paper is organized as follows. Section 2 reviews cognitive theories such as perceptual organization and segregation. Section 3 discusses how perceptual theories help us design and evaluate software tools. The last section summarizes this paper.

2. Cognitive theory

In this section, we present several fields in cognitive science, such as the theory of perception and perceptual organization, which may help us to better evaluate tool support.

2.1. Perceptual theory

It is widely accepted that visualization helps people learn things, but how the brain processes, transforms, and interprets visual stimuli is still unclear. Various theories of perception have been proposed [12]:

Marr's theory [8]: Cognitive functions are filters that operate on raw visual stimuli and turn them into information.

Gibson's theory [5]: We shift our attention based on the the structure of the environment and create a cognitive map to interact with the world.

Gestalt theory [2, 10]: We restructure our perception in ways that make it unified and coherent. This theory formulates the principles of organization which explain why some displays are better than others. Perceptual organization will be discussed in more detail in the next subsection.

Theory of notation [12]: Many researchers try to find good notations for visualization and those notations include symbol systems to create graphs that convey design semantics.

2.2. Perceptual organization

Perceptual organization is referred to how separated and segregated objects in the world that we perceive are located and interact with one another [3]. Research in perceptual organization studies how small elements are grouped into larger objects [6], which is important for tool interface designers to design humane and interactive user interfaces [4, 14]. The following are several important principles of perceptual organization, most of which are from the Gestalt Laws [3, 6].

Prägnanz: Prägnanz is a German word which means "suggestive figure". Therefore, the law of prägnanz is also called the law of good figure or the law of simplicity. That is, the patterns are perceived in such a way that their structures are as simple as possible.

Similarity: Similar elements (e.g., shape or color) appear to be grouped together.

Continuation: The law of good continuation states that points belong together if they result in straight or smoothly curve lines when connected, and lines are grouped together in such a way as to follow the smoothest path.

Proximity or Nearness: Elements that are close to each other will be grouped together.

Closure: We tend to complete incomplete figures to fill in gaps and units, forming a closed figure that tends to be perceived as a whole.

Common Fate: Things that are moving in the same direction will be grouped together.

Familiarity: According to the law of familiarity, or meaningfulness, things are more likely to be grouped together if the groups seem familiar or meaningful.

Synchrony: The law of synchrony states that "visual events that occur at the same time will be perceived as going together".

2.3. Perceptual segregation

Compared with perceptual organization, research in perceptual segregation basically studies the problem of "figure-ground segregation" to indicate when objects are separated [6]. Objects are usually referred to as figures and the background is called the ground. The fundamental figure-ground theory was proposed by Gestalt psychologists who believed that the figure is more like things that we are familiar with and it is more memorable than the ground. They also stated that the figure is perceived as being in front of the ground, and the contour of the figure and the ground more likely belongs to the figure. Experiments show that the following factors make an object more like a figure:

- Symmetric areas are usually seen as a figure.
- Stimuli with relatively smaller areas tend to be seen as a figure.
- Vertical or horizontal orientations have higher probabilities to be seen as a figure than other orientations.
- Meaningful and familiar objects are more likely to be seen as a figure.

Modern theory about figure-ground segregation discovered that contours (i.e. edges) play a very important role in figure-ground perception [7].

The laws of perception explain how our visual system identifies objects and how we put together the basic features to observe a coherent, organized world of things and surfaces. From the software visualization perspective, the principles of perceptual organization provide the basic design rules to organize multiple artifacts so that the users can group related information easily and without ambiguity. The figure-ground theory suggests what kind of user interface would be clear and suggestive.

3. Applying perceptual theories

In this section, we demonstrate how perceptual theories can be applied to evaluate tools and provide design ideas through SHriMP (Simple Hierarchical Multi-Perspective), an application and technique for visualizing and exploring information spaces in multiple perspectives [1, 9]. Figure 1 shows the main user interface of SHriMP, displaying the software structure of a popular game called "Hangman".

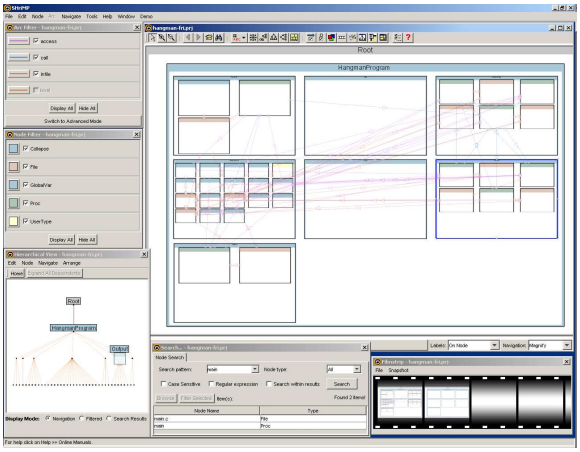


Figure 1. The User Interface of SHriMP

3.1. Gestalt theory

The Gestalt theory supports that "the whole, in perception, is more than the sum of its parts" [3, 16]. This statement implies that the integration of visual information within a tool is very important to support cognition. SHriMP nicely integrates the view of a software system into one nested graph, which avoids the ambiguity of switching between different windows. See the main view in Figure 1.

SHriMP groups artifacts based on a system hierarchy in its main view, i.e. child nodes will be grouped together and displayed inside their parent node. However, grouping artifacts from the same level is a little ambiguous in SHriMP. Figure 2 shows the global variables of the "Hangman Program". You might notice that all the artifacts are similar and within the same distance to each other. The arcs connecting the artifacts are not very clear and they often cross other artifacts. There are no perceptual organization laws that can be applied directly to explain how these artifacts should be grouped. Therefore, users will probably get confused by this visualization window.

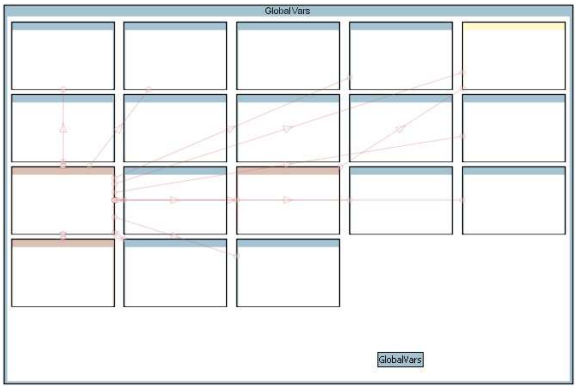


Figure 2. The global variable window

3.2. Similarity, proximity, and familiarity

SHriMP tries to solve the problem of grouping by providing different layouts. Figure 3 shows four different kinds of representations of the global variable window: (a) radial layout; (b) spring layout; (c) vertical tree layout, and (d) horizontal tree layout. Thus, different perceptual organization principles such as *proximity*, *continuation*, and *familiarity* can be applied to group these artifacts. For example, artifacts in the radial layout can be grouped based on their proximity (nearness). We can use the continuation and familiarity laws to group artifacts in the tree layout.

Although different layouts can reduce the perceptual ambiguity, there are still difficulties for the user to group artifacts that are close to each other or connected. SHriMP uses different colors on the title bar of artifacts to separate them by type, but due to color blindness, some people might not be able to distinguish these different artifacts [6]. Even for normal people, colors in a small title bar are still not very suggestive. Form perception theory indicates that shapes can be used to separate objects [3].

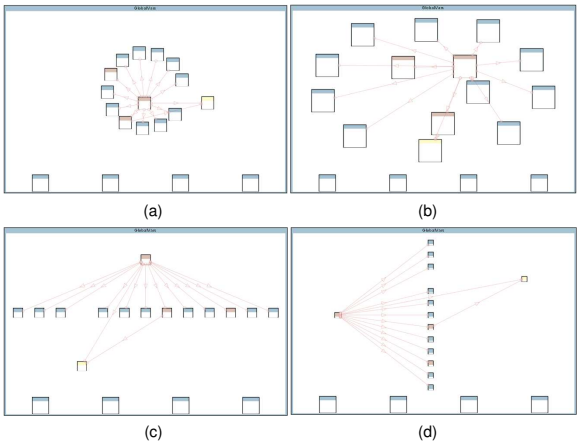


Figure 3. Different layouts of the global variable window

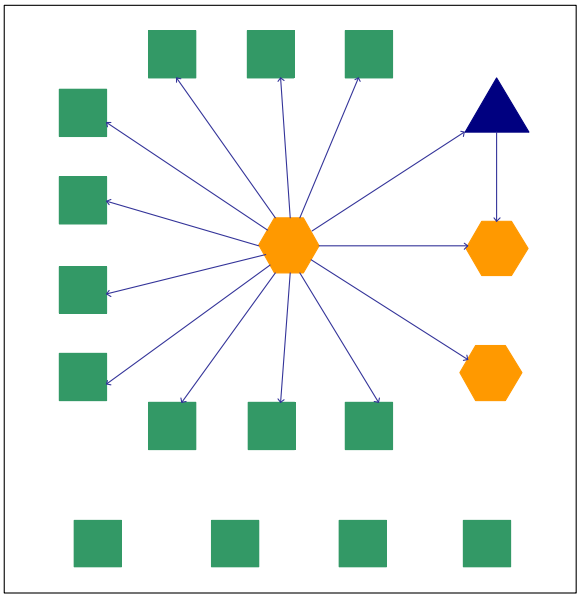


Figure 4. Using different shapes to represent the radial layout of the global variable window

We propose representing different types of artifacts as

different shapes: Figure 4 shows our representation of the radial layout and Figure 5 shows the horizontal layout. Different perceptual organization laws such as *similarity*, *proximity*, and *familiarity* can be applied to group objects in this figure.

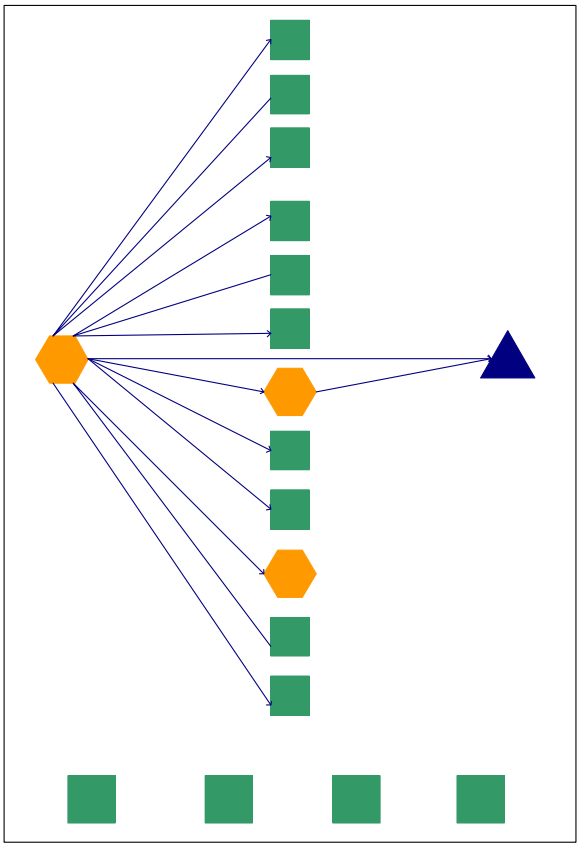


Figure 5. Using different shapes to represent the horizontal tree layout of the global variable window

3.3. Common fate and synchrony

The laws of *common fate* and *synchrony* are valuable for designing animations in software visualization tools. Some animations in SHriMP follow the common fate principle. For example, when the user changes the layout of a window from the default layout to the radial layout, the artifacts that will be grouped together move in the same direction. This may be one of the reasons why users appreciate this kind of animation. The filtering function in SHriMP follows the synchrony principle: when the user hides and shows a certain type of node (artifact), the same group of nodes will appear or disappear together. But, this kind of filtering can only be applied by type. For the future, it would be nice if the user can show and hide arbitrary groups of nodes so that these groups can be perceived more clearly.

3.4. Figure and ground

According to perceptual segregation, the visualization can be categorized into two parts: objects (figure) and background (ground). SHriMP's default layout has a "figure-ground" defect. In SHriMP, a parent node can be perceived as ground and its child nodes as figures. The law is that stimuli with smaller areas tend to be seen as figure and small elements tend to be perceived as larger figures when they are arranged in groups. Therefore, there should be enough space on the ground for the figures [16]. However, in SHriMP's default layout, the ground is a little small and the figures are tight, which makes the figure-ground effect unclear, as shown in Figure 6 (a). It would be better if the default view looks like Figure 6 (b).

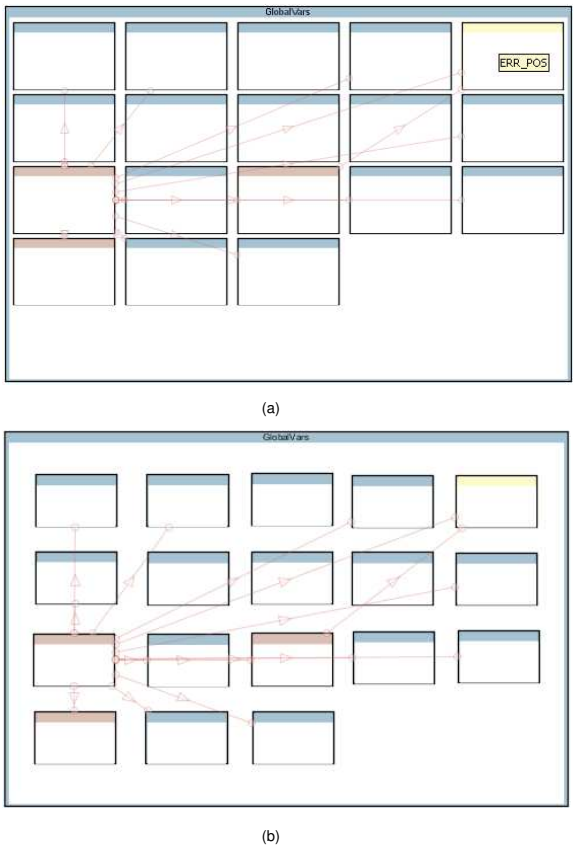


Figure 6. Figure and ground in SHriMP view

From Petre's user study of programmer's mental imagery [13], we can conclude that much of design is non-verbal and visualization is essential. However, the main is-

sue is that how to identify the most appropriate visualization for a given problem or domain. Future software visualization tools should provide specific visualizations for a given domain. We believe that a visualization tool that can provide different kinds of views for different problem domains will be better adopted.

4. Summary and future work

Cognitive theory can help us design and evaluate software visualization tools. In this paper, we presented some cognitive theories which would be helpful to understand and improve tool adoption. We evaluated SHriMP based on perceptual principles. The key ideas of this paper are:

- Cognitive theory helps us to discover the nature of how programmers understand programs.
- Combining cognitive science and software engineering principles would guide tool designers in finding the appropriate cognitive support.
- By applying perceptual organization laws, we evaluated SHriMP and proposed some suggestions to improve the tool, such as presenting artifacts by different shapes, showing/hiding a portion of nodes, and leaving enough background space for artifacts.

For future work, we would like to further explore cognitive science theories and apply them to improve tool adoption. We plan to run both quantitative and qualitative experiments [15] to verify our assumptions and theoretical analysis, especially on how cognitive theories would improve tool adoption rate.

5. Acknowledgement

We would like to thank Walter F. Bischof for his valuable suggestions on reviewing perceptual theories.

References

[1] SHriMP (Simple Hierarchical Multi-Perspective) website. http://shrimp.cs.uvic.ca/.

[2] J. G. Benjafield. *Cognition*. Prentice Hall, 1992.

[3] K. R. Boff, L. Kaufman, and J. P. Thomas. *Handbook of Perception and Human Performance. Volume II. Cognitive Processes and Performance*. Wiley-Interscience Publication, 1986.

[4] A. Dix, J. Finley, G. Abowd, and R. Beale. *Human-Computer Interaction*. Prentice-Hall, Inc., 2th edition, 1998.

[5] J. J. Gibson. *The Ecological Approach to Visual Perception*. Boston: Houghton Mifflin, 1979.

[6] B. E. Goldstein. *Sensation and Perception*. Wadsworth-Thomson Learning, 6th edition, 2002.

[7] R. Kimchi, M. Behrmann, and C. R. Olson. *Perceptual Organization in Vision: Behavioral and Neural Perspectives*. Lawrence Erlbaum Assoc Inc, 2003.

[8] D. Marr. *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information*. W H Freeman, 1982.

[9] J. Michaud, M.-A. D. Storey, and H. A. Müller. Integrating information sources for visualizing Java programs. In *Proceedings of the International Conference of Software Maintenance (ICSM 2002)*, 2001.

[10] P. Moore and C. Fitz. Gestalt theory and instructional design. *Journal of Technical Writing and Communication*, 23(2):137–157, 1993.

[11] H. A. Müller, J. H. Jahnke, D. B. Smith, M.-A. Storey, S. R. Tilley, and K. Wong. Reverse engineering: a roadmap. *22nd International Conference on the Future of Software Engineering*, pages 49–60, 2000.

[12] M. Petre, A. Blackwell, and T. Green. Cognitive questions in software visualisation. *Invited chapter in: Stasko, J., Domingue, J., Brown, M., and Price, B. (Eds.), Software Visualization: Programming as a Multimedia Experience. MIT Press. 453-480*, 1998.

[13] M. Petre and A. F. Blackwell. A glimpse of expert programmers' mental imagery. In *Papers presented at the seventh workshop on Empirical studies of programmers*, pages 109–123. ACM Press, 1997.

[14] J. Raskin. *The Humane Interface: New Directions for Designing Interactive Systems*. Addison-Wesley, 2000.

[15] J. Shaughnessy, J. Zechmeister, and E. Zechmeister. *Research Methods in Psychology*. Higher Education, 6th edition, 2002.

[16] M.-A. D. Storey. Information visualization and knowledge management course website, 2003, University of Victoria. http://www.cs.uvic.ca/~mstorey/teaching/infovis/.

[17] M.-A. D. Storey, F. D. Fracchia, and H. A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Systems and Software*, 44(3):171–185, 1999.

[18] M.-A. D. Storey, K. Wong, and H. A. Müller. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, 36(2-3):183–207, 2000.

[19] A. Walenstein. *Cognitive Support in Software Engineering Tools: A Distributed Cognition Framework*. PhD thesis, School of Computing Science, Simon Fraser University, May 2002.

[20] A. Walenstein. Improving adoptability by preserving, leveraging, and adding cognitive support to existing tools and environments. In *Proceedings of the 3rd International Workshop on Adoption-Centric Software Engineering*, pages 36–41, 2003.

[21] C. Ware. *Information Visualization: Perception for Design*. Morgan Kaufmann Publishers, 2000.

[22] I. Zayour and T. C. Lethbridge. Adoption of reverse engineering tools: A cognitive perspective and methodology. In *Proceedings of Ninth International Workshop on Program Comprehension*. IEEE Press, 2001.

Lessons learnt in adopting automated standards enforcement tool

Bala Brahmam Dasoji
Tata consultancy services
bala.dasoji@tcs.com

Abstract

In this position paper we describe the challenges faced in adoption of a Standard checking tool by taking the case study of a programming standards checking tool ASSENT developed at Tata Research and Development and Design Center, Pune, India. In spite of the technical capability of the tool and depth of program analysis as compared to other tools in the market, it was found that the acceptance rate among the users was low. Usability was identified as one of the key areas which need improvement. In this paper we describe some of the usability issues, the remedial measures taken to address these and the lessons learnt in the process.

1. Introduction

Identifying defects early in the life cycle of software development is important for every application. A good software development process would require programs to adhere to well defined programming standard. Such adherence can be verified using code reviews. One common problem occurring with code reviews is that there is a lack of focus as reviewers try to address multiple goals – functionality, modularity, reusability, adherence to standards etc. An automatic standards checker can be of great value to programmers as it can ensure the completeness and consistency with respect to laid down standards and allows reviewers to focus on other aspects of code review. [1] discusses some of the challenges faced in adopting ASSENT a standards checking tool that was built at Tata Research Design and Development Center, Pune, India. The key issues raised in [1] are

- Large number of non-conformances reported
- Definition of programming standards
- Severity Levels
- False negatives
- Integration into process
- Performance
- Language dialects
- Modify/Customize rules

In this position paper we extend the discussion to the usability related issues with the tool. The reasons for low acceptance rate of the tool were analyzed and usability of tool was identified as one of the major problem areas. A number of steps were taken to address some of the usability issues. In the process a number of lessons were learnt that could help other tool development projects. The remainder of the paper is organized as follows: section 2 discusses the various usability related issues faced with the tool, section 3 describes the remedial measures taken to overcome the problems, section 4 discusses the lessons learnt in the process and section 5 discusses the future enhancements to the tool.

2. Usability Issues

2.1. Customization

As mentioned in [1], every project typically customizes corporate or industry standards to meet its own requirements. To facilitate this, ASSENT provides a library of rules. The user chooses the rules appropriate for the project and checks the code against this subset of rules. It might so happen that the existing library of rules may not be sufficient to meet all the coding standard needs. The required rule may not either exist in the library or customizations may be required.

Due to the above requirement users wanted the feature in ASSENT where they could add their own rules, modify existing rules.

2.2. Team needs

A project is a team effort. Once it is decided that a particular set of standards will be followed, all the members of the team should follow the same set of

standards. ASSENT had only one mode of installation in which the user was given full control to choose the set of rules which will be checked. This freedom also came with the side effect that each team member can now choose which set of rules to follow the freedom of the team member to choose rules to be adhered to be is the source of inconsistency in standards followed across the team. External co-ordination among the team members was required in order to ensure that all the members of the team followed the same set of standards This situation gave rise to a requirement where only the team leader decides which set of rules will be followed for the project where as the team members use the set of rules decided by the team leader and are not allowed to make any modifications to the pre decided set.

2.3. Multiple configurations

ASSENT provided its users the facility to select and deselect the rules from the library that will be enforced. This set of rules selected forms a configuration for this project. If some changes are made to this configuration the original changes would be lost and new configuration would become active. There was no way of saving a particular configuration for later use. Users who are performing standards check on code with different requirements for standards had to remember which set of rules were to be followed for which code. This dependency on user's remembrance gave rise to the requirement of multiple configurations and ability to save configurations for later use.

2.4. Development Environments

Most of the users of ASSENT Java used one of the available Integrated Development Environments (IDE) in the market for developing their applications. Usage of IDE significantly speeds up the development process and improves productivity. ASSENT was shipped as a stand alone tool so the users had to export files and other settings from their IDE to ASSENT and this was tedious process many times they missed settings like CLASSPATH which the IDE used to take care of automatically for them. Users wanted a way where they could run ASSENT standards checker on their code without leaving their IDE. It is reasonable enough for them to have such a requirement because they were able to perform all the tasks related to their application like editing, compiling, deploying without leaving their IDE they would also want to check their code for standards without leaving their development environment.

2.5. Auto-correction

ASSENT checks for programming standard violations and reports back to user in an interactive browser that marks the particular section of the code that caused violation of the rule. The tool provides facilities where non-conformances can be browsed source-file wise, rule wise and rule group wise. The tool also produces non-conformance reports in html and MS Excel formats. Many users wanted a feature where ASSENT along with reporting the violation also auto-correct the code so as to remove the violation upon users consent.

2.6. Other languages

Developers who have used ASSENT in their projects for checking standards of their code written in C and JAVA came back to us with the requirement of a similar tool for other languages used in their projects like C++, COBOL, C#, VB etc.

3. Remedial measures

3.1. Customization

ASSENT uses a language called VERLANG (VERification LANGuage) for writing rules. Writing a new rule or modification of existing rule requires the knowledge of VERLANG and the internal representation of language constructs. The learning curve required to attain this knowledge is approximately 2 months. A user who wants to add his own rules obviously can not spend so much time in learning VERLANG and language construct representation to write a few custom rules required for his project. Providing the user with ability to write new rules or modify existing rules is difficult in general. We have taken the following steps to solve this problem partially. If the user requirement for a new rule is general enough to be included in the library the ASSENT team takes up the task of writing the new rule and adding this rule to ASSENT library this process is time consuming. For adding a new rule to the library approximately 2 person days of effort was required. After adding new rules required in the above fashion for some time we figured out that most of the custom rules required by users were related to naming conventions. We have also observed that some of the new rules added to the library were very similar to the existing rules and just

differed in single/multiple constants which could even be provided at run time. Based on this observation we went ahead with the concept of parameterization of the rules.

In this feature some rules of the library are treated specially and they can take parameters. These parameters are provided by the user while configuring the rules based on the requirements of the project. 30% of the rules in ASSENT rule library are now parameterized and take one or more parameters from the user. Naming convention rules were taken up as a special case of parameterization where the whole consists of only parameters provided by the user. Some efforts were put into the design of a framework where the end user will be able to write new rules the complexity involved in designing such a framework was huge. The stumbling block in designing such a framework was the fact that user is unaware of the internal representation of language constructs which are the building blocks for ASSENT rule library. Requests for new rules which do not fit the parameterization framework are handled by ASSENT team. Examples of parameterized rules and Naming convention rules are given below.

Avoid large methods

The definition of large in the above rule differs from project to project. Some users might want to set the limit to 50 lines where as some others might want it to be 75 lines. By parameterization of the rule we allow the user to specify the limit on the number of lines.

All Classes should start with capital letter

To define such a naming convention rule the user selects "Class" as the rule entity and specifies a constraint in the form of the regular expression "[A-Z][A-Za-z0-9]*" using the naming convention rules GUI provided with the tool. The tool then checks all the entities of type "Class" against the regular expression provided. A violation is reported if the given name does not match the regular expression.

3.3. Team needs

To meet the requirement of configuration by team leader and that configuration to be used by all the other team members we have introduced 2 modes of installation, administrator installation and user installation. Administrator installation will allow enabling and disabling of rules, addition/modification/removal of rule parameters and naming convention rules. A user installation does not allow any of the above changes. The above limitation allows the team leader to prepare a configuration of rules from the library and enforce all the team members to use the same configuration ensuring consistency in the standards followed.

3.4. Multiple configurations

For enabling the use of multiple configurations facilities have been added in the tool where the user can select/deselect a particular set of rules from the library and then save this configuration with an easy to remember name. This configuration can then be used later for performing standards check. Such a saved configuration can also be shared with other members of the team.

3.5. Development Environments

Most of the Java IDE's available in market today are closed in nature i.e. they don't allow integration of third party tools easily. We have also surveyed users of ASSENT regarding the IDE that they use. From the survey results we found that Eclipse and WSAD were the widely used IDEs among ASSENT users. Eclipse is an open source IDE with plug-in centric architecture and WSAD is based on Eclipse technology. Due to the plug-in centric architecture of Eclipse it easily allowed third party tools to be plugged into the IDE and promised seamless integration even the most basic features of IDE like compiler and editor were provided as plug-ins. ASSENT team went ahead to write ASSENT plug-in for eclipse and provided users with a facility where they could perform standards checking on their code without leaving their development environment. The process of Standards checking was now as easy as compiling source files. Non-conformances were reported along with compiler errors and warnings. Since WSAD was based on eclipse technology ASSENT plug-in for eclipse also works directly on WSAD without any changes.

3.6. Auto correction

ASSENT works on the intermediate representation of program source. If some code was auto-corrected that may have impact at many other places in the application. Accurately replacing all such impacted code is difficult to achieve ASSENT team is currently working on a solution to this problem. As a first step towards auto correction ASSENT C has a facility where alternative solutions are presented to the user and he applies one of them manually. Alternative solutions was not of much benefit to the users because the changes still need to be done manually.

3.7. Other languages

To build ASSENT for a new language most of the effort goes into building a language parser and designing intermediate representation for the language. If the expected user base is not big enough, spending huge effort on building a new parser is not feasible. Efforts were made to build ASSENT for C#. The effort for building a C# parser was huge so it was discontinued.

4. Lessons Learnt

1. A language without dialects is very easy to support. C has many dialects. ASSENT MISRA-C was aimed at embedded system projects but hardly any embedded C code was written in ANSI-C each embedded C compiler came with its own extensions and ASSENT failed to parse such code with extensions. Unlike C Java doesn't have dialects and follows a single language specification. The parser for Java is very robust and ASSENT did not have any problems parsing any Java code.
2. Auto correction was the most sought after feature but techniques for auto correction are still in nascent stage. Much effort needs to be devoted for developing techniques for auto-correcting code that does not conform to standards. Auto correction also needs strong analysis of code regarding the impact caused by changing a piece of code. In some cases the impact may not be so evident and outside the scope of the tool. Example: Changing the name of a public member of a public class has impact that is sometimes outside the scope of application.
3. Some enhancements like parameterization and customized naming convention rules have found good acceptance among users where as some others like alternative solutions were not successful.
4. The core features of standards checking and depth of analysis in ASSENT have not changed in the past one year. Just by addressing the usability related issues we have found that the internal usage of tool has increased from 200 to 1000 users. The first impression that a user has on the tool is mostly determined by the user friendliness of the tool. User will go ahead and test the technical capabilities of the tool only when it is easy to use and all the actions to be performed are intuitive. This suggests that apart from the scalability challenges posed by static program analysis, usability related issues should never be undermined.

5. What next?

The planned future enhancements to the ASSENT tool include the following
1. Variety of graphical reports where a summary of the non-conformances can be seen in the form of pie-charts, bar graphs etc.
2. Software metrics for the code analyzed.
3. Providing powerful GUI which enables user to write rules easily.
4. Focused research on auto correction.

6. Conclusions

We have explored the possible usability related issues in a standard checking tool by taking a case study of ASSENT standards checking tool. The various usability related issues faced by users of the tool were discussed along with suitable solutions offered by the tool. We have also looked at the lessons learnt in the process some of which could be applied to most of the tools. In the case of ASSENT improving features related to usability have resulted in better adoption of the tool. Apart from the technical capabilities, user friendliness and usability should also be given high priority to improve the adoption of the tools.

7. References

[1] T K Shivaprasad, Vipul Shah. Challenges faced in adopting automated standards enforcement tools. ACSE 2003

[2] Ashok Sreenivas and Vipul Shah. Assent: An Automatic Specification-driven Standards Enforcement Tool. Technical report, Tata Research Development and Design Centre, Pune, India, 2000 - 01.

[3] H. Pande and A. Sreenivas. Darpan: A program analysis framework. Technical report, Tata Research Development and Design Centre, Pune, India, 1998.

[4] M.S. Hecht. Flow Analysis of Computer Programs. Elsevier, North-Holland, 1977.

On Goals and Codes in Distributed System

----An Explanation of the Concepts of Perfect Ball

Zhou Zhiying
Tsinghua University, Beijing, China
zgzy-dcs@tsinghua.edu.cn

Abstract

The concepts of perfect ball indicate that the goals of the project for distributed systems are related to the uncertainty. "Position" and "Momentum" cannot be determined simultaneously in the microcosms based on the duality of the wave-particle. The uncertainty relation of distributed systems, as the main concepts of perfect ball, addresses that the codes, as available products, and the goals, as the announced features of the software products, cannot be determined simultaneously. The change rates, the "factorial-dimension" as the special case of fractal dimension, the well designed experiments, and the fine social and living study are part of the main concerns for further study on the concepts of "perfect ball".

1. Introduction

This paper discusses goals and codes in distributed system as well as concepts of perfect ball [1] for better understanding software development projects for distributed systems.

The term of perfect ball with some kind of modern happiness style was initiated from Heisenberg's uncertainty relation and ancient oriental philosophies.

The fundamental concepts of perfect ball are the duality of the thing. The objective facts in software development are explicit and controllable, but the implied subjective facts and unknown or hidden information are uncontrollable and unavoidable. The duality of the thing results uncertainty.

The next section discusses the relation between goals and codes. The third section lists part of the main concerns related to the concepts of perfect ball.

2. Goals and Codes

Firstly let's recall the goal space of the project. Usually, the successful projects and/or end-products are driven from the controlled finite goals of the project. It corresponds to the separable entities or the controllable finite variables of the black boxes, and referred to the goals space with multi-dimension. The goal of the project is formally referred to the pre-fixed or planned points in the goal space of the project.

We call the perfect point as the reachable goal, which is the pre-fixed point or the dynamic point in the goal space. The arguments about increasing the time and efforts for better quality are true to a monolithic system with the limited scope and the stable environments. But it is questionable in the reality of distributed systems, whose features are characterized as uncertainty, fuzzy boundaries, and the complexity with almost unlimited facts.

Therefore, perfect point is the target of the project traditionally. But it cannot be true for the complex distributed systems. The "goals" are hardly referred to the planned points in the goal space because the existed uncertainty.

The developers of software project try to control the development processes of defining, implementing, modifying and validating activities during the system life cycle, and finally to reach the project goal. There is a fuzzy misunderstanding about goals. People usually regard the end product (all the source codes, executable codes, and documents are simply named as codes in this paper) as equivalent to goals because they think that validation processes and verification processes are ideal and successful. But the goals relate with the behaviors of the system, and the codes are the systems themselves. They are different thing.

We know that the development efforts are only directed to the codes, but indirectly to the goals of the project. Furthermore, the indirectness usually implies a dangerous misleading. Please refer to figure 1 and consider thoughts, addresses and objects with their relations among the distributed developers, artifacts and users. It brings more serious issues.

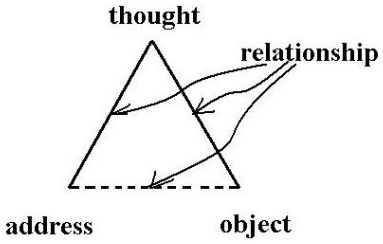


Figure 1 Triangle among thought, address, object and their relations

According to Heisenberg' uncertainty relation, "position" and "momentum" cannot be determined simultaneously in the microcosms, which is completely different with the macrocosms. We may ask if the difference between the distributed systems and traditional monolithic system is similar to the difference between classical mechanics and quantum mechanics. Is there uncertainty relation of distributed system? The following mappings will help us to uncover the interesting.

Let's imagine the first mapping between the microcosms and the complex distributed systems. If codes in the complex distributed project correspond to the role of the position of the particle in the microcosms, the goals or requirements of system behaviors will correspond to the role of the momentum. The uncertainty relation of distributed systems, as one of main concept of perfect ball, addresses that the codes, as available products, and the goals, as the announced features of software products, cannot be determined simultaneously.

Therefore we may say that it is almost impossible to define the exact and complete features of the complex distributed system from the known codes, and it is also hard to produce the codes as the end product with the exact pre-defined features, especially if there are considerable changing influences from the outside.

The second mapping is also very interesting. The particle-wave (also position-momentum) corresponds to visible-invisible in the microcosms. In the distributed system, the codes are the visible as the texts located at the special positions in the mediums of the software products, and the behavior or goals as the effects of code executing, are invisible in the global perspective.

Software development engineering and re-engineering are the processes from the goals/codes to the codes/goals. Actually both directions feature some non-inverse, which against the traditional fundamentals of the formal methods. The concepts of perfect ball reflect the current change and evolution trends of software development methodologies, from mostly pursuing ideal mechanics to the reality of companying the social and living. It can be regarded as the stage beyond the steps from scratches to rigor rules in the traditional development concepts.

3. The current and future researches on perfect ball

The current and future researches on perfect ball include the experimental and theoretical works. The following lists the few of issues with brief discussions.

- When and in what area the concepts of perfect ball are effective? The answer might come from rates of changes during developing distributed systems.
- How to define the perfect ball formally? The parameters in Heisenberg uncertainty relation are related to the experiments in the microcosms with the velocity of light. It brings us to think the importance of designing good experiments relating to the concepts of perfect ball.
- What are the metrics related to the concepts of perfect ball? A promising tool might be named as "factorial-dimension": the special cases of the fractal dimension [2, 3]. It might reduce the difficulty in multi-dimension viewpoints of software architecture of the distributed system from infinitely increasing the number of the dimension of the traditional geometry. The innovative metrics will also be helpful to MDA, model driven architecture, and meta-model [4].
- What benefits could come from the concepts of perfect ball? At least, it provides the better psychology factors, which help to reduce the unnecessary pressures of pursuing the unreachable absolute perfect. The unnecessary pressures would push people to work on unnecessary laborious tasks for engaging in the unreachable perfect, or pursuing the false perfect ever forgetting the true imperfect, as commercial advertisements with some non scientific characters. It helps software organizations and developers to remove their ideal perfect dreams and move back to the reality. It also uncovers the need to renew software test concepts, such as those in test driven method. A case study of developing the prototype [5] was discussed for exploring the advantages.

In addition, further researches on perfect ball should introduce the social and living study in software development methodologies, instead of only emphasizing the pure logic of formal methods of constructing the rigor rules as the traditional mechanics nature. So far, Taiji Model [6] has explored the facility of inserting the ancient Chinese philosophies as well as uncertain theory in the software technologies; and the open source developments [7, 8, 9] allow the flexible goals for the different developers and users. All those are the interesting approaches for our study.

Acknowledgements

The author would like to thank Mr. Rao Talasila and Prof. Fan Yan for the helpfulness to English grammar. A special thank to Dr. Dong Yuan at Department of Computer Science and Technology, Tsinghua University for exchanging the viewpoints.

References

[1] Zhou Zhiying, The perfect ball in software development, International workshop on Software development methodologies of distributed systems, Amsterdam, The Netherlands, Sept. 19-21, 2003, Proceedings of SDM-DS 2003, Beijing, China, 2004. (TBA)

[2] Peter Coveney and Roger Higfield, The Arrow of Time, W.H.Allen, London UK, 1990.

[3] Benoit B. Mandelbrot, The fractal geometry of nature, W.H. Freeman and Company, New York, U.S.A. 1983.

[4] Object Management Group, http://www.omg.org.

[5] Shi Li, Zhou Zhiying, Case study: A Prototype Development by Test-driven Method, International workshop on Software development methodologies of distributed systems, Beijing China, Dec. 16-18 2003, Proceedings of SDM-DS 2003, Beijing, China, 2004. (TBA)

[6] Zhou Zhiying, CMM in uncertainty environments, Communication of ACM, August 2003, p. 115-119.

[7] Coskun Bayrak and Chad Davis, The relationship between Distributed system and Open Source Development, Communication of ACM, Vol.46, No.12, 2003, p. 99-102.

[8] Eric.Raymond, The Cathedral and the Bazaar, http://www.tuxedo.org/~esr/writings /cathedralbazaar.

[9] Open Source Initiative, The Open Source Definition-Version 1.14, http://www.open source.org /docs/definition.hmtl, 2004.

Adoption Centric Problems in the Context of System of Systems Interoperability

Dennis Smith, Edwin Morris, David Carney

Carnegie Mellon University

Software Engineering Institute

Abstract

A critical emerging software need concerns interoperability between systems and systems of systems.To meet this increasing demand, organizations are attempting to migrate existing individual systems that employ disparate, poorly related, and sometimes conflicting systems to more cohesive systems that produce timely, enterprise-wise data that are then made available to the appropriate users.

This paper first identifies a set of adoption centric principles that need to be addressed to obtain interoperability. It then outlines a model to facilitate better interoperability. The model distinguishes between programmatic interoperability, constructive interoperability and operational interoperability. The paper finally outlines a set of issues that need to be addressed in order to facilitate system of system interoperability.

1 Background

A critical emerging software need concerns interoperability between systems and systems of systems. This need is emerging within just about every software domain, including ebusiness, egovernment, integration of large military systems, mergers and acquisitions, and communications between embedded devices of systems that are traditionally considered to be hardware, such as automobiles and aircraft.

To meet this increasing demand, organizations are attempting to migrate existing individual systems that employ disparate, poorly related, and sometimes conflicting systems to more cohesive systems that produce timely, enterprise-wise data that are then made available to the appropriate users. Meeting this goal has often proven to be considerably difficult. This paper will focus on adoption centric issues that need to be addressed to obtain interoperability.

2 Background

A recent SEI study focused on identifying current problems in interoperability within DoD systems [Levine 02]. All of the problems were collected from interviews and workshops with personnel from the DoD.

The study indicated that new systems designed and constructed to interoperate with existing and other new systems, and adhering to common standards, still fail to interoperate as expected. Several of the general problems related to a need to have a better understanding of issues related to more effective adoption. These include:

- incomplete requirements
- unexpected interactions, and
- unshared assumptions

Specific interoperability problems included:

- Planned interoperability between new systems is often scaled back in order to maintain compatibility with older systems that cannot be upgraded without major rework.
- Strict specification of standards proves insufficient for achieving desired levels of interoperability, because organizations constructing “compliant” systems interpret specifications in different ways, thus creating different variants of the links.
- Policies promote a single point of view at the expense of other points of view. For example, policies that enhance the levels of interoperability that can be achieved in one domain are generalized to additional domains, where they unduly constrain organizations trying to produce interoperable systems.
- Funding and control structures in general do not provide the incentives necessary to achieve interoperability.
- Tests constructed to verify interoperability frequently fail to identify interoperability shortfalls. In other cases, systems are

approved for release in spite of failing interoperability tests.

- Even when interoperability is achieved by systems of systems, it is difficult to maintain as new versions of constituent systems are released. New system versions frequently break interoperability.

3 Guiding Principles

As Fred Brooks pointed out more than 15 years ago, the factors that make building software inherently difficult are complexity, conformity, changeability, and invisibility [Brooks 87]. With apologies to Brooks, we assert that achieving and maintaining interoperability between systems is also inherently difficult, due to:

- complexity of the individual systems and of the potential interactions between systems
- lack of conformity between human institutions involved in the software process and resulting lack of consistency in the systems they produce
- changeability of the expectations placed on systems (particularly software) and the resulting volatility in the interactions
- invisibility of all of the details within and between interoperating systems

In spite of a considerable effort, technical innovations aimed at improving software engineering have not successfully attacked the problems represented by these essential characteristics. In fact, today's interoperating systems are likely more complex (due to the massive increase in the number of potential system-of-systems states) than those examined by Brooks. They exhibit less conformity (due to the increased diversity of the institutions involved in construction of the constituent parts), are more volatile (due to the need to accommodate widely diverse users) and have even poorer visibility (due to size, number of participating organizations, etc.)

We therefore posit a set of six principles that will inform our efforts in the selection of problems to address and, more critically, in the analysis of potential solutions. The principles are

1. No clear distinction exists between *systems* and *systems of systems.*
2. Most interoperability problems are independent of domain.
3. Solutions cannot depend on complete information.
4. No one-time solution is possible.
5. New technologies constantly move systems toward legacy status.
6. Networks of systems demonstrate emergent properties.

3.1.1 No Clear Distinction Between Systems and Systems of Systems

The distinction between a system and a *system of systems* is often unclear and seldom useful. By this we mean that many, perhaps a majority, of "systems" are actually systems of systems in their own right. The critical factor is less where a boundary might lie and more where control lies: most systems are now created with some components over which the integrator has less than complete control. Further, most systems must cooperate with other systems over which the integrator often has no control.

It is often stated that "What someone considers to be a system of systems, somebody else considers a system." Thus, for any given entity, one's perspective could see it as a component (of a larger system), as a system in itself, or as a system of systems. The Raptor's avionics system is certainly "a system." And, more importantly, *there usually is no top level*, because inevitably there will be some demand to include any system of systems in a more encompassing system of systems.

3.1.2 Interoperability Problems Independent of Domain

Most complex systems in almost every domain are now expected to interact with other complex systems. Regardless of domain, interoperability problems persist, and the costs of failures are huge. As an example, within the U.S. auto supply chain, one estimate put the cost of imperfect interoperability at one billion U.S. dollars per year, with the largest component of that cost due to mitigating problems by repairing or reentering data manually [Brunnermeier 99].

Our expectations are for even greater degrees of interoperability in the future, a goal that may prove difficult to achieve. The current generation of interoperable systems at least tend to be knowledgeable participants in the interaction—that is, the systems are being designed (or modified) specifically to interact with a particular system (or limited set of systems) in a controlled manner, and to achieve predetermined goals. What is new about

the future generations of interoperating systems is an emphasis on dynamically reconfigurable systems. These systems—or more accurately the services they provide—are expected to interoperate in potentially unplanned ways to meet unforeseen goals or threats.

We do not suggest that the solutions eventually found for the interoperability problems should be identical across domains. But we believe that the various communities should be aware of each other and look for commonality of high-level purpose and solution strategy—if not of solution detail—within other communities.

3.1.3 Solutions Cannot Rely on Complete Information

Classic software engineering practice assumes a priori understanding of the system being built, including complete and precise comprehension of

- assumptions or preconditions expected of the system that are required for successful use, including standards, system and environmental conditions, and data and interactions expected of other hardware, software, and users
- functionality, services, data, and interactions to be obtained from and provided to outside agents
- non-functional properties or quality of service required by the system and expected of the system from interacting components

For interoperable systems, the same information is required by all participants: the individual components (i.e., the individual systems), the links between them, and the composite system of systems. It would therefore seem that for an organization building a component (system), complete knowledge of all expectations is necessary to complete it. Unfortunately, we seldom (if ever) have such complete and precise specification even *when* a single system is *only* expected to operate in isolation.

The reality is that multiple organizations responsible for integrating multiple systems into interoperating systems of systems have multiple—and rarely parallel—sets of expectations about the constituent parts, as well as different expectations about the entire system of systems. The decisions that they make about the overall system of systems, e.g., assumptions, preconditions, functionality, and quality of service, are just as likely to be as incomplete and imprecise as those of organizations responsible for a single system.

Given that having complete and precise information about a system of systems (and its constituent parts) is not possible, two approaches to managing the potential chaos are evident:

- Reduce imprecision by enforcing common requirements, standards, and managerial control.
- Accept imprecision as a given and apply engineering techniques that are intended to increase precision over time, such as prototyping and spiral models of development.

The first approach alone may well increase interoperability to a significant degree, but it is also highly static and does not address the inherent imprecision in the software engineering process or the legitimate variation in individual systems. The second approach is limited in a different way, since without agreeing on some level of commonality, we are left with an "every system for itself" world that will not approach the levels of interoperability we require.

3.1.4 No One-time Solution Is Possible

We live in a dynamic and competitive world in which the needed capabilities of systems must constantly change to provide additional benefits, to counter capabilities of adversaries, to exploit new technologies, or in reaction to increased understanding or evolving desires or preferences of users. Simply put, systems must evolve to remain useful.

This evolution affects both individual systems and systems of systems. Individual systems must be modified to address unique and changing demands of their specific context and users. The expectations that systems of systems place on constituent systems will likewise change with new demands. However, the changing demands placed on a system by its immediate owners and those placed by aggregate systems of systems in which it participates are often not the same, and in some cases are incompatible.

The result is that maintaining interoperability is an ongoing problem. This was verified by our interviews with experts who had worked with interoperability. In some cases, desired system upgrades did not happen because of the impending effect on related systems. In other cases, expensive

(often emergency) fixes and upgrades were forced on systems by changes to other systems.

In order to maintain interoperability, new approaches are needed to

- vet proposed requirements changes at the system and system-of-systems level
- analyze the effect of proposed requirements and structural changes to systems and systems of systems
- structure systems and systems of systems to avoid (or at least delay) the effect of changes
- verify interoperability expectations to avoid surprises when systems are deployed

New approaches to structuring systems that anticipate changes, that vet requirements and structural changes and analyze their impact, and that verify that systems of systems perform as anticipated will go a long way toward maintaining the interoperability of related systems.

3.1.5 Networks of Interoperability Demonstrate Emergent Properties

Emergent properties are those properties of a whole that are different from, and not predictable from, the cumulative properties of the entities that make up the whole. In very large networks, it is not possible to predict the behavior of the whole network from the properties of individual nodes. Such networks are composed of large numbers of widely varied components (hosts, routers, links, users, etc.) that interact in complex ways with each other, and whose behavior "emerges" from the complex set of interactions that occur.

Of necessity, each participant in such real-world systems (both the actor in the network and the engineer who constructed it) acts primarily in his or her own best interest. As a result, perceptions of system-wide requirements are interpreted and implemented differently by various participants, and local needs often conflict with overall system goals. Although collective behavior is governed by control structures (e.g., in the case of the networks, network protocols), central control can never be fully effective in managing complex, large-scale, distributed, or networked systems.

The net effect is that the global properties, capabilities, and services of the system as a whole emerge from the cumulative effects of the actions and interactions of the individual participants propagated throughout the system. The resulting collective behavior of the complex network shows emergent properties that arise out of the interactions between the participants.

The effect of emergent properties can be profound. In the best cases, the properties can provide unanticipated benefits to users. In the worst cases, emergent properties can detract from overall capability. In all cases, emergent properties make predictions about behavior such as reliability, performance, and security suspect.[1] This is potentially the greatest risk to wide-scale networked systems of systems. ISIS recognizes that any long-term solution must involve better understanding and managing of emergent properties.

4 A Conceptual Approach to Begin Addressing the Interoperability Problem

To address the problem of interoperability, the SOSI model of interoperability has recently been proposed [Levine, 2003]. This model distinguishes between three types of interoperability

1) programmatic interoperability, which encompasses the activities related to the management of one program in the context of other programs

2) constructive interoperability which addresses technologies that create and maintain interoperable systems, and

3) operational interoperability which addresses the activities related to the operation of a system in the context of other systems.

These three types of interoperability are each briefly discussed in the following subsections.

[1] There is a continuing debate as to whether a property is truly emergent or whether, as we learn more about the components and their interactions, we will begin to find techniques for prediction of properties that appear to be emergent. We are neutral with regards to this debate, but believe that we must begin to understand the nature of emergent properties in networked systems of systems and find ways to manage problematic behaviors that occur—whether we can predict them or not.

4.1 Programmatic Interoperability

While programmatic interoperability requires the management of one program in the context of other programs, in reality most programs are managed in isolation, with little attempt to consider the management functions of other systems. Interoperability between related systems is typically defined by a common specification. However, because of the limitations of specifications and the lack of incentives for programs to probe beyond them, the resulting interoperability is often less than desired.

To remedy this situation, management approaches and techniques that bridge the gaps between the isolated programs and perspectives are needed. Some candidate strategies for achieving programmatic interoperability include

- synchronization of schedules and budgets
- joint risk management
- coupled award fee boards
- linked promotions

Achieving programmatic interoperability in a system-of-systems context demands consistent collaboration among the various program offices. Communication must be consistent and risk must be shared and distributed across programs. Failure to build management structures that encourage programs to interoperate will only perpetuate stovepipe management practices.

4.2 Constructive Interoperability

Constructive interoperability technologies commonly include shared architectural elements, data specifications, communications protocols, and common standards. Constructive interoperability is the nuts and bolts of what we commonly think of as system and software engineering. We expect good engineers to select the right technologies and approaches and apply them in thoughtful ways.

Selecting and using some common set of technologies is necessary, but is normally not sufficient to produce highly coupled interoperation. Currently available technologies capture only part of rich semantic context and underlying assumptions regarding data that are required for sophisticated interoperation. For example, our research examined two systems intending to interoperate that used object request brokers provided by different COTS vendors. The two program offices assumed that conformance to an industry standard would ensure interoperability. Unfortunately, the vendor of one broker added unique features to the product that extended the standard. These unique features were used during construction of one of the systems, making it impossible for the two systems to interoperate without considerable rework to one or both systems.

Achieving constructive interoperability demands new perspectives on the use of standards and software architectures. It is naïve to assume that simply using a standard will guarantee system interoperability. Joint definition of the standards and system-of-systems architecture will provide a critical aspect for each system's construction.

4.3 Operational Interoperability

Operational interoperability activities include defining

- Assumptions, business rules and doctrine governing the way the system is used
- conventions for how the user interprets information derived from interoperating systems (i.e., the semantics of interoperation)
- mechanisms for interacting with program offices to improve interoperability between programs

Problems in achieving interoperability at other levels inevitably lead to problems at the operational level. For example, in one case, multiple combat platforms were intended to exchange data. However, the platforms supported different communication links. Each type of link places different restrictions on the data. As a result, different users were receiving different views of the battle environment. This problem at the constructive level of interoperability also led to operational conventions for *how* the user interpreted data.

4.4 The Interoperability Environment

The model we posit is still incomplete, in that it does not address another set of actors who are critical to the interoperability problem and solution. These actors—particularly important within the DoD enterprise but with analogs in commercial enterprises as well—are involved in establishing high-level vision, setting policy direction, and creating or selecting enterprise-wide

standards. An expansion of the model includes these actors and their activities as part of the environment in which interoperability must be achieved and maintained (Figure 1).

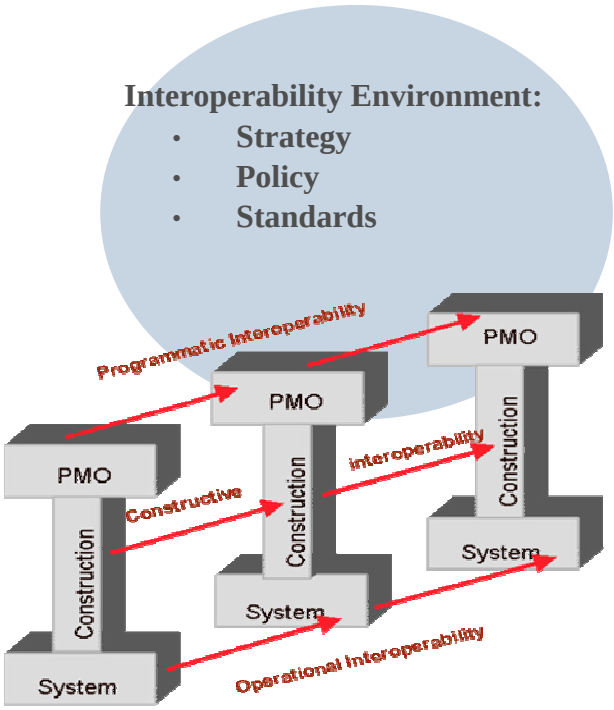


Figure 1: The Interoperability Environment

It is a mistake to think of the environment in which interoperability must be achieved as a single entity. In reality, this environment is made of many organizations that contribute high-level vision, policy, and standards. Unfortunately, according to the SOSI experts, these contributions can conflict and at times hinder efforts to achieve interoperability. In effect, there may be a lack of interoperability between various visions, policies, and standards.

4.5 Contributions of the SOSI model

The contribution of the SOSI model to the discourse is seen in the emphasis on *activities that must be performed by various actors* in order to achieve the interoperability goal. In a sense, it is an attempt to create a bridge between the interoperability problem (achieving interoperability at some expected level) and solution (presumably involving improved practices and technologies). The SOSI model identifies four major groups that will have to be part of any eventual solution:

- leadership that is responsible for creating strategy, setting policy, and establishing standards
- program offices responsible for management of interoperating systems
- organizations that select architectures and technologies and implement the systems
- operational entities that participate in the definition of requirements, and eventual use, of the system of systems

5 Conclusion

The SEI's ISIS initiative seeks to provide a focal point for sharing information and new technologies about interoperability among many scattered communities of interest. During the coming years, we expect that work will be done, both by us and others in the software engineering community, to

- codify current management and engineering practices for construction and evolution of systems of systems
- identify and characterize techniques for forming and evolving systems of systems
- develop evaluation approaches for selecting constituent elements of systems of systems
- analyze current and emerging technologies and products with potential applicability to the integration and interoperability of systems of systems
- develop cost models

References

Brooks, Fred. "No Silver Bullet: Essence and Accidents of Software Engineering." *IEEE Computer 20*, 4 (April 1987): 10-19.

Brunnermeier, Smita B. & Martin, Sheila A. *Interoperability Cost Analysis of the U.S. Automotive Supply Chain*. National Institute of Standards & Technology, March 1999. <http://www.nist.gov/director/prog-ofc/report99-1.pdf>.

Levine, L.; Meyers, C.; Morris, E.; Place, P.; & Plakosh, D. *Proceedings of the System of Systems Interoperability Workshop (February 2003)* (CMU/SEI-2003-TN-016). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. <http://www.sei.cmu.edu/publications/documents/03.reports/03tn016.html>.

Data and State Synchronicity Problems While Integrating COTS Software into Systems

Alexander Egyed, Sven Johann, and Robert Balzer
Teknowledge Corporation
4640 Admiralty Way, Suite 1010
Marina Del Rey, CA 90292, USA
{aegyed,balzer}@teknowledge.com

Abstract

The cooperation between commercial-off-the-shelf (COTS) software and in-house software within larger software systems is becoming increasingly desirable. Unfortunately, COTS software packages typically are standalone applications that do not provide information about changes to their data and/or state to the rest of the system. Furthermore, they typically use their own, proprietary data formats, which are syntactically and semantically different from the system.

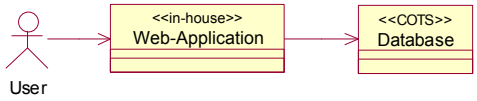
We developed an approach that externalizes data and state changes of COTS software instantly and incrementally. The approach uses a combination of instrumentation and reasoning to determine when and where changes happen inside the COTS software. It then notifies interested third parties of these changes. The approach is most useful in domains where the system has to be aware of data and state changes within COTS software but it is computationally infeasible to poll it periodically (i.e., large-scale data). The approach has been validated on several COTS design tools with large, industrial models (e.g., over 34,000 model elements) for effectiveness and scalability.

1. Introduction

Incorporating COTS software into software systems is a desirable albeit difficult challenge. It is desirable because COTS software typically represents large, reliable software that is inexpensive to buy. It is challenging because software integrators have to live with an almost complete lack of control over the COTS software products [2].

We define a COTS-based system to be a software system that includes COTS software [3]. From a software architecture perspective, a software system consists of a set of interacting software components. We thus also refer to COTS software used within a COTS-based system as COTS components. A COTS-based system may include one or more COTS components among the set of its software components.

Incorporating COTS software into new and existing COTS-based systems has found strong and widespread acceptance in software development [1]. For example, a very common integration case is in building web-based technologies on well-understood and accepted COTS web servers. Indeed, COTS integration is so well accepted in this domain that virtually no web designer would consider building a web server or a database anew.



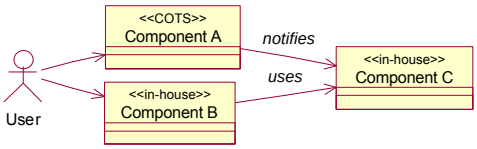
While there are many success stories that point to the seaming ease of COTS integration there are also many failures. We believe that many failures are the result of not understanding the role the COTS software is supposed to play in the software system. In other words, the main reason of failure is architectural.

Many software systems, such as the web application above, use the COTS software as a back-end; its use is primarily that of a service-providing component. Only some form of programmatic interface (API) is required for COTS software to support this use. COTS software vendors have become increasingly good at providing programmatic interfaces to data and services of their products.

COTS integration is less trivial if the COTS software becomes (part of) the front end; e.g., with its native user interface exposed and available to the user. These cases are rather complex because the COTS software may undergo user-induced changes (through its native user interface) that are not readily observable through the programmatic interface. In other words, the challenge of COTS integration is in maintaining the state (e.g. data model, configuration, etc) of the COTS software

consistent with the overall state of the system even while users manipulate the system through the COTS native user interface.

This paper discusses this issue from the perspective of using COTS design tools such as IBM Rational Rose, Mathwork's Matlab, or Microsoft PowerPoint. These design tools are well-accepted COTS software products. They exhibit commonly understood, graphical user interfaces. This paper shows how to use these COTS products, with their accepted user interfaces, to build a functioning software system where the architectural integration problem is more like this:



2. Data Consistency and State Synchronicity

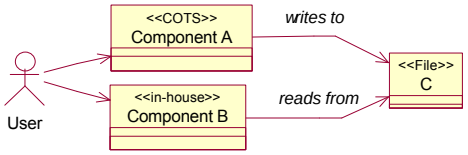
Most COTS software products initiate interaction with other software products they are explicitly designed to interact with. This is problematic because there are integration scenarios where COTS software is required to interact with software it was not explicitly designed to interact with. Moreover, while it is true in some cases that COTS software is fully proactive (with respect to changes to its internal state and data), we found that COTS software with user-driven GUIs (graphical user interface) tend to be less proactive. This raises the severe problem of maintaining the consistency and synchronicity of shared data and state between such a COTS component and its system while a user is manipulating it. The challenges are:

1) Data Inconsistency: Data captured in COTS software may be consumed by other components in a system. If a user manipulates the data within the COTS software then this may introduce inconsistency in the shared data. The problem is that COTS software typically does not know or care about notifying other components of internal changes.
2) State Synchronicity: User actions in COTS software may have system relevance in some cases. COTS software does not understand the needs of a system it is part of and consequently does not recognize user actions the system must be notified about. System relevant user actions may get lost if they are done through the native user interface of the COTS software.

In an ideal world, COTS software is configurable to notify other components of relevant internal changes (data and state). In such an ideal world, the COTS software becomes an active participant in the COTS-based system into which it is being integrated. Today it is rare for COTS software to have these capabilities built-in. For integration, this creates a major challenge of how such COTS software can be augmented from the "outside" so that internal activities (state and data changes) relevant to other components are proactively communicated to them. The next sections discusses how this can be accomplished using a combination of instrumentation and reasoning.

3. Batch Change Detection

Batch notification is to externalize all relevant COTS state information (e.g., by exporting design data from Rational Rose) so that it becomes available to other components. This is probably the easiest solution but has one major drawback in that it is a loose form of incorporating COTS software into software systems. State information has to be replicated with the drawback that even minor state changes to the COTS software cause major synchronization activities (e.g., complete re-export of all design data from Rational Rose).



Batch notification is computationally expensive and only then feasible if the integration between the COTS software and the rest of the system is loose or the amount of data/state information is small.

The following will present our approach to incremental notification. Our approach wraps COTS software to identify changes as they occur with several beneficial side effects:

- Only changes are forwarded
- Changes are forwarded instantly
- Change detection is fully automated

4. Incremental Change Detection

To understand COTS changes one has to be aware of *when* and *where* the COTS software undergoes changes. This problem does not exist with in-house developed components because they can be programmed to notify other, affected components of a system. However, the lack of access to the source code makes this approach impossible for COTS software. Therefore, to incorporate COTS software into systems the COTS software must be observed to identify and forward all relevant events to the system.

4.1. The *When* and *Where* of Change Detection

There is a trivial albeit computationally infeasible approach to observing changes in COTS software by caching its state and continuously comparing its current state with the cached state. Unfortunately, this approach does not scale well if the COTS state consists of large amounts of data (e.g., design data in tools such as IBM Rational Rose, Mathwork's Matlab, or Microsoft PowerPoint). To better observe changes in COTS software it has to be understood *where* and *when* changes happen.

Take, for example, IBM Rational Rose. In Rose a user creates a new class element by clicking on a toolbar button ("new class") followed by clicking on some free space in the adjacent class diagram. A class icon appears on the diagram and the icon is initially marked as selected. By then clicking on the newly created, selected class icon once more, the name of the class may be changed from its default or class features such as methods and attributes may be added. These changes could also be done by double-clicking on the class icon to open a specification window. There are two patterns worth observing at this point:

- Changes happen in response to mouse and keyboard events only
- Changes happen to selected elements only

The first observation is critical in telling *when* changes happen. It is not necessary to perform (potentially computationally expensive) change detection while no user activity is observed. The second observation is critical in telling *where* changes happen (Egyed and Balzer 2001). It is not necessary to perform (potentially computationally expensive) change detection on the entire COTS design data (state) but only on the limited data that is selected at any given time.

Both observations are the key for scalable and reliable change detection in GUI-driven COTS software.

4.2. Caching and Comparison

Changes are detected by comparing a previous state of a COTS software with its current state. Generally, this implies a comparison of the previously cached state with the current one. Knowing the time when changes happen and the location where changes happen limits what and when to compare.

Basic Change Detection

After every mouse/keyboard event, we ask Rose what elements are selected (via its programmatic interface) to compare these elements with the ones we cached previously. If we find a difference (e.g. a changed name, a new method) between the cached elements and the selected ones then we notify other components (e.g., in-house developed components) about this difference. Thus, our approach notifies other components on the behalf of the COTS software. If we find a difference then we also update the cache to ensure that differences are reported once only. Obviously, the effort of finding changes is computationally cheap because a user tends to work with few design elements at any given time only.

Our approach uses the programmatic interface of the COTS software to elicit and cache state information. For example, in IBM Rational Rose this includes design data such as classes, relationships, etc. The caching is limited to "relevant" state information that is of interest to other components of the system. For example, if it is desired to integrate some class diagram analysis tool with Rose, then change detection may be limited to class diagrams only (i.e., ignoring sequence diagrams, state chart diagrams). In summary, basic change detection is as follows:

1. download and cache data when first selected
2. re-download data and compare with cached data when de-selected
3. update cache
4. no need to cache selected element a second time if it was selected previously because the cache stays up-to-date

This approach detects changes between the cached and the current state. However, there are two special cases: 1) new elements cannot be compared because they have never been cached and 2) deleted elements cannot be compared because they do not exist in the COTS software any more.

The creation and deletion problem can be addressed as follows. If we cannot find a cached element for a selected one then this implies that it was newly created (otherwise it would have been cached earlier[1]). We then notify other components of the newly created element and create a cached element for future comparison. In reverse, if an element in the cache does not exist in the COTS software then it was deleted. Other components are thus notified of this deletion and the cached element is deleted as well. Note that a deletion is only detected after de-selection (i.e., a deleted element is a previously selected element that was deleted) and a creation can only be detected after selection.

[1] Note: we pre-cache all data identifications initially to understand the difference between newly created data and old data that was never selected.

Ripple Effect of Change Detection

Until now, we claimed that changes happen to selected elements only. This is not always correct. Certain changes to selected elements may trigger changes to "adjacent", non-selected elements. For example, if a class X has a relationship to class Y then the deletion of class X also causes the deletion of the relationship between X and Y although the latter is never selected.

There are two ways of handling this ripple effect. The easiest way is to redefine selection to include all elements that are affected by a change. For example, if a class is selected then we define that all its relationships are selected also. Change detection then compares the class and its relationships. This approach works well if the ripple effect does not affect many adjacent elements (e.g., as in this example) but it could become computational expensive.

A harder but more efficient way of handling the ripple effect is to implement how changes in selected elements affect other, non-selected elements. For example, we implemented the knowledge that the deletion of a class requires its relationships to be deleted also. In this case, neither the creation of a class nor its change does have the same ripple effect.

Anomalies

We found that basic change detection and their ripple effects cover most scenarios for change detection. However, there are exceptions that cannot be handled in a disciplined manner. We found only few scenarios in Rose that had to be handled differently.

For example, the state machines in Rose have a peculiar bug in that it is possible to drag-and-drop them into different classes while the programmatic interface to Rose does not realize this. If, in the current version of Rose, a state machine is moved from class A to class B then, strangely, both classes A and B believe they own the state machine although only one of them does. We thus had to tweak our approach to also consider the qualified name of a state machine (a hierarchical identifier) to identify the correct response from Rose. Obviously, this solution is very specific to this anomaly. Fortunately, not many such anomalies exist.

5. Change Detection Infrastructure

Figure 1 depicts our infrastructure for augmenting COTS software schematically [5]. The center of the figure holds the actual COTS software. Since no source code is available, it cannot be changed from within. *Instrumentation* [6] is used to monitor outside stimuli directed towards the COTS software (shaded frame around the COTS software). For example, we use instrumented wrapper technology to observe interactions between a software component and its environment (e.g., mouse and keyboard events). A customized *Reasoning* component within the framework then uses information made available through instrumentation and from inspection of the COTS component's state and data (via its programmatic interface) to infer what internal changes this activity caused. It essentially implements Caching and Comparison discussed above.

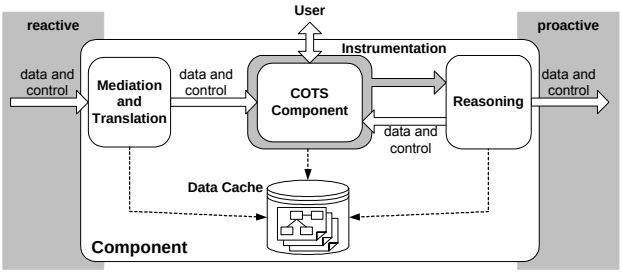


Figure 1. Augmenting a COTS Component from the Outside through Mediation, Translation, Instrumentation, and Reasoning [5]

Mediation and translation is used to construct alternative interfaces for COTS products to facilitate their use as components in larger systems. Mediators and translators [6] augment native interfaces of COTS software (i.e., wrappers or glue code). The purpose of translation is to make COTS-specific data and control information available in a format that is understood by other components of the COTS-based system (e.g., to impose a standardized interface on a COTS software). The purpose of mediation is to bridge middleware platforms (e.g., COM [15], CORBA [9,13], DLL, RMI [11]) between COTS software and other components of the system. Other components of that system then do not use the COTS native interface directly but instead use the new, augmented interface. Using an augmented interface has the advantage that the appearance of COTS software is "altered" without changing the COTS software itself (see also Figure 1). The services of the COTS software are then provided in the alternative format without other components and the COTS software being aware of this.

The optional data store is required to save cached data and other information.

6. Validation

The integration framework was validated on three major COTS products of different vendors (IBM's Rational Rose, Mathwork's Matlab/Stateflow, and Microsoft PowerPoint). Each COTS product was consequently integrated with different in-house and third-party systems. In total, over ten integration case studies were performed that tested the validity of our approach. For example, Rose was integrated with the UML/Analyzer

system for automated consistency checking between UML class diagrams and C2 architectural descriptions [12]; it was integrated with an automated class diagrams abstraction software [4], the SDS Simulator for executing UML-like class and statechart diagrams [7], the Boeing/MoBIES Translator and Exporter for modeling embedded systems [10], and several other systems. Similarly, Matlab/Stateflow and Microsoft PowerPoint were integrated into yet other systems like the Design Editor for modeling user-definable notations [8] or the survey authoring system [14].

Scalability is key to our approach. The largest model our approach was tested on was an Avionics design model from Boeing with over 43,000 model elements. While the initial pre-caching takes about 20 seconds, the subsequent caching and comparison done with every mouse or keyboard event is unnoticeable to human users. Boeing engineers and other groups have used our change detection approach without scalability issues.

Although our case studies demonstrated a wide range of applicability of our integration infrastructure, it cannot be considered proof of its general applicability. To date, our focus was primarily on COTS software with graphical user interfaces that do externalize significant parts of their internal data. In the context of these systems, we have repeatedly demonstrated that it is possible to integrate COTS software in a scalable and reliable fashion. The quality of the COTS-based systems was evaluated through numerous scalability and usability tests. To date, our infrastructure has been used by several companies (e.g., Boeing, Honeywell, and SoHaR) and universities (e.g., Carnegie Mellon University, University of Southern California, Western Michigan University).

7. Approximation

It is generally easier to maintain consistency between COTS software and the system it is being integrated with if the semantics of the COTS data is similar to the semantics of the system data. For example, we integrated Rational Rose design information with UML-compatible design information and both are conceptually similar. Unfortunately, consistency becomes more complicated if the data of COTS software is re-interpreted into a semantically different domain. This is not uncommon. For example, many applications exist that use Rose as a drawing tool. In those cases, the meaning of boxes and arrows may differ widely.

This section discusses how to “relax” change detection depending on the difficulty of the integration problem. This problem was motivated by our need to having a domain-specific component model, called the ESCM (Embedded Systems Component Model) [10], integrated with Rational Rose. While it is out of the scope to discuss the ESCM, it must be noted that its elements do not readily map one-to-one to Rose elements. As such, there are cases where the creation of an element in Rose may cause deletions in ESCM and there are cases where overlapping structures in Rose may relate to individual ESCM elements. This integration scenario is problematic because it is very elaborate to define how changes in Rose affect the ESCM.

Previously, we solved the integration problem by comparing Rose data with cached data. User actions, such as mouse and keyboard events, triggered partial re-transformations to compare the current Rose state with the cached copy. The comparison itself was trivial; so was up-date. The key was transformation.

The main difficulty of integrating the ESCM is in determining what to re-transform and what to compare. This is a scoping problem and it becomes more severe the more complex the relationship between system data (e.g., ESCM) and COTS data becomes. While we implemented a very precise, incremental change notification mechanism for Rose->ESCM (its discussion is out of the scope), we found that it is often good enough to approximate change detection.

Thus, we simplified the problem by implementing change detecting with the possibility of reporting false positives (Rose change is reported that does not change the ESCM) but the guarantee of not omitting true positives (Rose change that changes the ESCM). In case of integrating ESCM with Rose, it was not problematic to err on the side of reporting changes that actually did not happen since they only led to some unnecessary but harmless synchronization tasks. The ability to relax the quality of change detection (i.e., false positives) strongly improved computational complexity in this case.

8. Conclusion

Consistency between commercial-off-the-shelf software (COTS), their wrappers, and other components is a pre-condition for many COTS-based systems. Our experience is that it is possible to observe data and state changes in GUI-driven COTS software even if the COTS software vendor did not provide a (complete) programmatic interface for doing so. This paper discussed several strategies for adding change detection mechanisms to COTS software.

Acknowledgement

We wish to acknowledge Neil Goldman, Marcelo Tallis, and David Wile for their feedback and comments.

References

[1] Boehm, B., Port, D., Yang, Y., Bhuta, J., and Abts, C. Composable Process Elements for Developing COTS-Based Applications. 2002.

[2] Boehm, B.W., Abts, C., Brown, A. W., et al: Software Cost Estimation with COCOMO II. New Jersey, Prentice Hall, 2000.

[3] Brownsword L. , Oberndorf P., and Sledge C.: Developing New Processes for COTS-Based Systems. *IEEE Software*, 2000, 48-55.

[4] Egyed A.: Automated Abstraction of Class Diagrams. *ACM Transaction on Software Engineering and Methodology (TOSEM)* 11(4), 2002, 449-491.

[5] Egyed, A. and Balzer, R.: "Unfriendly COTS Integration - Instrumentation and Interfaces for Improved Plugability," *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE)*, San Diego, USA, November 2001.

[6] Egyed A., Medvidovic N., and Gacek C.: A Component-Based Perspective on Software Mismatch Detection and Resolution. *IEE Proceedings Software* 147(6), 2000, 225-236.

[7] Egyed, A. and Wile, D.: "Statechart Simulator for Modeling Architectural Dynamics," *Proceedings of the 2nd Working International Conference on Software Architecture (WICSA)*, August 2001, pp.87-96.

[8] Goldman, N. and Balzer, R.: "The ISI Visual Editor Generator," *Proceedings of the IEEE Symposium on Visual Languages*, Tokyo, Japan, 1999, pp.20-27.

[9] Object Management Group: The Common Object Request Broker: Architecture and Specification. 1995.

[10] Schulte, Mark. MoBIES Application Component Library Interface for the Model-Based Integration of Embedded Software Weapon System Open Experimental Platform. 2002.

[11] Sun Microsystems: Java Remote Method Invocation - Distributed Computing for Java. 2001.(UnPub)

[12] Taylor R. N., Medvidovic N., Anderson K. N., Whitehead E. J. Jr., Robbins J. E., Nies K. A., Oreizy P., and Dubrow D. L.: A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering* 22(6), 1996, 390-406.

[13] Vinoski S.: CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications Magazine*, 1997.

[14] Wile D.: Supporting the DSL Spectrum. Journal of Computing and Information Technology. *Journal on Computing and Information Technology* 9(4), 2001, 263-287.

[15] Williams S. and Kindel C.: The Component Object Model: A Technical Overview. *Dr. Dobb's Journal*, 1994.

Challenges Posed by Adoption Issues from a Bioinformatics Point of View

Daniel L. Moise Kenny Wong Gabriela Moise
Department of Computing Science
University of Alberta, Canada
E-mail: {moise,kenw,gabi}@cs.ualberta.ca

Abstract

Developing interoperability models for data is a crucial factor for the adoption of research tools within industry. In this paper, we discuss efficient data interoperability models within a field where they are highly needed: the bioinformatics field. We present the challenges that interoperability models for data must face within this field and we discuss some existing strategies built to address these challenges. The potential of a semi-structured data model based on XML is discussed. Also, a novel approach that enhances the capabilities of the data integration model by automatically identifying XML documents generated based on the same DTD is presented. Practices developed within this application domain can be used for the benefit of similar adoption issues in various other domains.

1. Introduction

One important direction in software engineering research is the development of efficient techniques for integrating existing and new research tools within the industrial setting. As noted in [11], this is a challenging task due to the ”adoption” problems that arise from the inherent differences between the software development process within the academic environment as opposed to the industrial environment. Overcoming the challenges posed by the adoption task would be of great benefit to both the research and the industrial world. Software production could achieve increased quality by exploiting cutting-edge research results. Researchers, on the other hand, can validate and improve the tools that they have developed often as a ”proof of concept” solution.

Previous work on software adoption issues classify the challenges faced by the adoption process as ”user-to-tool” compatibility concerns and ”tool-to-tool” compatibility concerns. Examples of adoption-related problems classified under the first category are primitive, user-unfriendly interfaces, lack of proper documentation or limited support for complex tasks. The second category of concerns motivates the development of mechanisms for data, control, and presentation integration. As hypothesized by the ACRE project [1], for a successful adoption of research tools by the industrial development processes, we need to enhance the compatibility of existing and new research tools with both existing users and other tools.

One domain in great need of effective adoption techniques and strategies is the domain of bioinformatics, which has experienced a striking development for the past decade. The gap between the academic and the industrial environments is doubled by the gap imposed by the dual nature of the field. Addressing the challenging problems of the bioinformatics domain requires solid computer knowledge and deep understanding of the underlying biology. Any software solution developed within this application domain should be both computationally and biologically effective, and importantly, easy to work with, extend or reuse.

Unfortunately, this is not always the case, as it has been observed by [5]. More than often, computer scientists develop complicated computational models that require a significant understanding effort on the side of the biologists. As a side effect, biologists hesitate to adopt new software tools, once they have some software tool that produces reasonable solutions. This problem can be alleviated by developing biologically-aware software solutions. This desideratum requires both a tight collaboration between the two fields, as well as the integration of the abundant biological data and applications that have accompanied the recent development of the bioinformatics field.

In this paper, we motivate the need for efficient adoption techniques with respect to the bioinformatics field. We discuss the challenges that interoperability models for data must face within this field and we present the existing strategies for integration of data and software solutions. We discuss the potential of a semi-structured data model through the use of XML [13] and XML schema [14]. Within this context, we propose an approach that enhances the capabilities of the data integration model by automatically identifying XML documents generated based on the same DTD.

The paper is structured as follows. In section 2, we discuss the challenges faced by the integration of biological data and software applications. Section 3 presents existing strategies for dealing with the problem of integration. Section 4 discusses a XML-based data representation model and details our proposal concerning the enhancement of this model through a software solution. We summarize and suggest directions for future work in section 5.

2. Challenges

In this section we discuss the challenges that interoperability models for data must face within the field of bioinformatics.

A large number of biological data sets and software solutions have appeared concurrently with the explosion of available genome sequence information over the past few years. To understand and interpret this data at the genome level, we need software tools that allow us full, flexible query access to an integrated, up-to-date view of all related information, without regard to the location of this data or its format. The biological data sets are widely distributed, both across the World Wide Web and within individual organizations. This data is stored in a variety of storage formats, such as flat text files databases, relational databases, object-oriented databases, XML databases, etc.

All of the above constitute obstacles that any interoperability model for data has to deal with. Besides the fact that data is spread over multiple, heterogeneous databases, some of these databases are not easily queried (flat file sequences, web sites, BLAST alignments [2]) or not even easily parsed. Many data sources do not represent biological objects optimally for the kinds of queries that investigators typically want to pose. For instance, GenBank (annotated database of DNA sequences) [7] is sequence-centric, not gene-centric. SwissProt (annotated database of protein sequences) [10] is sequence-centric, rather than domain-centric. Finally, queries must operate on the most up-to-date versions of the data sources for obvious reasons.

3. Discussion of the Existing Approaches

This section presents existing strategies for dealing with the problem of integrating biological data.

Previous work on models for data integration within the bioinformatics field centered around two main strategies:

1. the consolidation approach, and
2. the federation approach

The *consolidation* approach constructs a single, homogeneous, mega database. Data sets of interest are collected together and translated into a common database structure. All the incompatibilities are ”scrubbed”, and the contributing legacy DBs are removed once and for all. This approach is rarely practiced and lacks feasibility, because with data growing at a fast rate, one time translation is not suitable. Also, solving semantic incompatibilities of contributing DBs and ”scrubbing” the rest of the data to fit this solution is a difficult, time and resources consuming process.

The *federation* approach has three variants:

1. Incorporate links within databases to one another
2. Couple DBs loosely
3. Data warehouse

In the first variant of a federation approach, data retrieval is done by traversal of links. No other requirements are imposed on the contributing DBs. However, this approach is prone to missing or inconsistent links. Also, there is no query facility to allow retrieval of multiple records based on user-specified criteria. Entrez ([6]) is an example of an integration system that is constructed on this strategy.

The underlying idea of the second federation approach is to construct queries over multiple DBs without touching the DBs themselves. For this purpose, a query processor is built that maps the pending query to each participant database and, afterwards, integrates answers together. The challenging task is the integration of the retrieved answers, which necessitates knowledge of the schema and semantics of the constituent DBs. OPM ([12]) is an example of a system built on this strategy.

Finally, the ”data warehouse“ approach develops a global schema for all the data in all the DBs. Data is transformed into this common schema and loaded in a central repository on a regular basis. Query facilities are provided by the central repository. However, there is a need to update the global schema when local DBs change their schemas or when new DBs arrive. This approach is similar to some extent to the consolidation approach, but in the case of a data warehouse, the constituent DBs still exist and are synchronized in the global DB.

An innovative approach to the problem of integration of biological data sets has been recently proposed by BioMOBY. BioMOBY [4] is an open source research project which aims to generate an architecture for the discovery and distribution of biological data through web services: data and services are decentralized, but the availability of these resources, and the instructions for interacting with them, are registered in a central location called MOBY central. Native BioMOBY objects are lightweight XML. They are used both as the query and the response of a SOAP transaction between the client and the service provider. Object and service ontologies, located at the MOBY central are used both for querying and for registration/de-registration of services. From

the point of view of the developer, this solution for integration is very convenient and easy to work with. This approach is more expensive for the service providers that have to build SOAP servers and to provide as much information as possible, not just on their service, but also on other services that are related or may be of potential interest.

4. XML-based data model

In this section, we discuss the potential of a semi-structured data model for modeling data interoperability through the use of XML and XML schema.

The issue of which data models are most efficient for biological data interoperability has been a topic of discussion often addressed by previous work. Traditional relational models obscure to some extent the comprehension of the biological objects, i.e., biologists often find it difficult to understand a large number of tables. This problem is better treated by object-oriented models, but, object-oriented approaches have difficulties in adapting to design changes and schema evolution [3], which may be reasonable requirements within the fast-changing environment of bioinformatics.

The XML standard has the potential of addressing some of the problems of data interoperability within the bioinformatics community. XML has several main advantages:

- It is flexible, extensible and reusable. Essentially, any change can be operated through the modification of the underlying DTD or XML Schema.
- It supports easy integration of the information from different data sources. As we have discussed above, the XML standard has been already used extensively within the bioinformatics community for information changes between data sources (e.g., BioMOBY).
- It can be easily read and parsed.

Building a data interoperability model based on XML and XML Schema as the underlying representation of the data has its own difficulties. A number of issues need to be addressed such as representing schemas of different data sources, schema information update as changes occur at the data source, etc. Within this context, we propose an approach that enhances the capabilities of the data integration model by automatically identifying XML documents likely generated from the same DTD. In the following, we detail the proposed approach.

4.1. Structural similarity of XML data

As the number of XML files increases dramatically, how to store and retrieve XML data becomes an urgent issue. One approach to solve the problem is grouping ”similar” XML files together based on their structures so that they can be stored and retrieved efficiently, especially in a heterogeneous environment, like the bioinformatics domain.

One important objective of grouping similar XML files is for efficiently indexing XML. Without grouping structurally similar XML files together, the objects for XML indexing may be scattered at different locations in the storage device. Thus, the performance will degrade significantly. Another application of XML similarity is automatically extracting a DTD for a set of XML documents. If we can cluster structurally similar XML documents, we can extract more specific DTDs for these documents. Such DTDs would facilitate more efficient searching by only accessing the relevant portions of data.

The challenge is how to define the similarity between XML files, i.e. what kind of metric should be applied. Several existing approaches leverage on the DOM tree representation of XML documents. Thus, measuring the structural similarity between two XML documents is equivalent to measuring the structural similarity between two trees. Existing algorithms devised for solving this problem define a set of allowed operations that can be applied for transforming one tree into another. The key idea is to find the ”cheapest” sequence of such operations, where each allowed operation has a cost associated with it. An example of a dynamic programming algorithm that follows this approach for measuring the similarity between XML documents is given in [8]. We note that the tree representation is not the only representation of XML documents used by previous research. For instance, in [9], the structure of an XML document is encoded as a time series, and the similarity between two XML documents is computed based on Fourier transformations.

We propose a novel approach to measure the structural similarity among XML data. We consider two XML documents to be structurally similar if they likely share the same DTD. We first introduce the data model used to represent XML documents.

An XML document can be represented as a labeled tree [8], where a node in the tree represents the corresponding element in the XML document and is labeled with the element’s tag name. We use the term ”tree” to refer to the labeled tree defined above, not the XML DOM tree. We ignore the value of each element since we are only interested in the structure of XML files. We represent the XML structure as a *structural tree*, which is a subtree of the XML labeled tree we have defined above, satisfying the condition that at every level there is no duplication of nodes. For example, the XML file in Figure 1 is represented as the structural tree in Figure 2. We represent an XML document as a set of paths as follows. A *root path* is a sequence of nodes starting from the root and ending at a leaf in the XML structural tree. Nodes are separated by

```
<organism>
        <sequence>
                <id>AY0700397.1</id>
                <length>961</length>
        </sequence>
        <sequence>
                <id>AY0710453.1</id>
                <string>aaacga...</string>
        </sequence>
</organism>
```

Figure 1. Example XML file

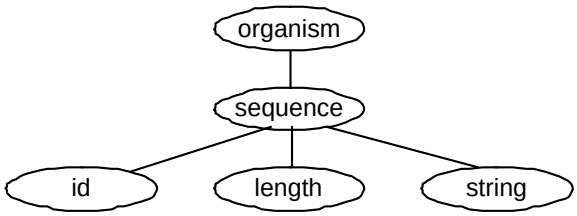


Figure 2. XML Structural Tree

the "/" character. For example, the root paths of the "organism" XML file are: "organism/sequence/id", "organism/sequence/length", and "organism/sequence/string". A *path* is a substring of a root path. For example, all paths of the root path "organism/sequence/id" are "organism", "organism/sequence", "organism/sequence/id", "sequence", "sequence/id" and "id". We use all the paths to represent an XML document.

Given a collection of XML documents, we represent each one of them as a set of paths, as we have explained above. We compute the frequency of each path; thus an XML document is now represented as a set of frequencies. Traditional research within the information retrieval field has proposed a large number of methods for measuring the similarity between two sets (or vectors) of frequencies. We have experimented with different such methods and our results have indicated that the cosine similarity measure performs the best.

Under this representation, we are able to compute the pair-wise distances among the XML documents of a given collection. The resulted distance matrix can be fed into any clustering algorithm, which will group similar XML documents together. In the context of our approach, "similar" XML documents translate into XML documents having "similar" structure. As indicated by our preliminary results, a large fraction of the XML documents that are clustered together have been generated based on the same DTD. These results suggest the potential of our approach for identifying XML documents likely generated from the same DTD.

This technique uses a simple, but effective, representation of the XML file as a set of paths. In addition, our method is significantly faster than the more complicated approaches based on tree matching.

Developing techniques such the one described above is useful for building more efficient interoperability models for data, which essentially means efficient solutions for addressing the "adoption" problem, tailored to the unique needs of the bioinformatics domain.

Bioinformatics is not the only application domain where efficient data interoperability models can be successfully applied. The practices developed here could virtually be applied to other instances of data integration problem with little modifications. Similar integration tasks have been the focus of active research in software engineering for many years. The strategies developed for bioinformatics purposes could be applied by software engineers to similar problems in various other application areas.

In addition, the bioinformatics domain offers many opportunities for learning how to make tools developed within academia effective in industrial practice. Research solutions need to quickly adapt to the continually changing demands of industry.

5. Conclusions

Aside from the traditional problems faced by adoption of research software within the industrial environment [1], the field of bioinformatics poses a number of extra challenges. As noted above, any software solution developed within this field has to be "biologically-aware" in order to be useful and usable. An important step towards biological awareness is finding efficient solutions to the problem of data integration. Exchanging data between information sources and analysis tools in a seamlessly manner is essential for building complex queries. We have shown that the integration task must face a series of specific challenges when exposed to the large, heterogeneous, distributed domain of biological data and applications.

We have discussed in this paper several models for data integration that have been proposed within the context of the bioinformatics domain. Without such models, the outcome of years of efforts of assiduous teams of researchers is at risk of remaining in shadow. Out of several potential models, XML-based data integration may be a feasible solution. To strengthen the capabilities of an XML-based integration model, we have proposed an approach for finding structurally similar XML documents. Our technique could be used to improve common tasks performed by a data integration model, such as efficient storing, indexing and querying of XML documents.

We suggest the investigation of other such techniques that have the potential to facilitate the construction of efficient data interoperability models. Building such models is an important task in itself, but it becomes even more important when projected to similar adoption problems in various other domains. Bioinformatics represents a challenging domain, but some of these challenges are encountered in other domains as well. Aside from solving important problems in bioinformatics, the technologies developed here could be applied in other application domains with little effort.

References

[1] Adoption-Centric Software Engineering Project, University of Victoria. http://www.acse.cs.uvic.ca.

[2] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 1990.

[3] J. Banerjee, W. Kim, H. Kim, and H. F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proceedings of the ACM-SIGMOD Annual Conference*, pages 311–322, 1987.

[4] BioMoby. http://www.biomoby.org.

[5] S. Davidson, C. Overton, and P. Buneman. Challenges in Integrating Biological Data Sources. In *Journal of Comptaional Biology*, 1995.

[6] Entrez. http://www.ncbi.nlm.nih.gov/entrez.

[7] GenBank. http://www.ncbi.nlm.nih.gov.

[8] H.V. Jagadish and A. Nierman. Evaluating structural similarity in XML documents. *Proceedings of the Fifth International Workshop on the Web and Databases (WebDB)*, 2002.

[9] S. Flesca, G. Manco, E. Masciari, L. Pontieri and A. Pugliese. Detecting structural similarities between XML documents. *Proceedings of the Fifth International Workshop on the Web and Databases (WebDB)*, 2002.

[10] SwissProt. http://us.expasy.org/sprot.

[11] S. Tilley, S. Huang, and T. Payne. On the Challenges of Adopting ROTS Software. In *3 rd International Workshop on Adoption-Centric Software Engineering, ACSE 2003*, pages 3–6, 2003.

[12] V.M. Markowitz and I. M. A. Chen and A. Kosky and E. Szeto. Opm: Object-protocol model data management tools '97. *Bioinformatics: Databases and Systems*, pages 187–199, 1999.

[13] XML. http://www.w3.org/xml.

[14] XML Schema. http://www.w3.org/xml/schema.

Leveraging XML Technologies in Developing Program Analysis Tools

Jonathan I. Maletic, Michael Collard, Huzefa Kagdi
Department of Computer Science
Kent State University
Kent Ohio 44242
330 672 9039
jmaletic@cs.kent.edu, collard@cs.kent.edu, hkagdi@cs.kent.edu

Abstract

XML technologies are quickly becoming ubiquitous within all aspects of computer and information sciences. Both industry and academics have accepted the XML standards and the large number of tools that support manipulation, transformation, querying, and storage of XML objects. Thus, tools and representations based on XML are very attractive with respect to adoption. This paper describes the experiences of the authors in the development and application of srcML, a XML application to support explicit markup of syntactic information within source code. Additionally, XML technologies are leveraged along with srcML to support various program analysis, fact extraction, and reverse engineering tasks. A short description of these tools is given along with the motivation behind using an adoption centric XML approach.

1. Introduction

While conducting research in program understanding, reverse engineering, and software visualization we ran into a common technical problem namely, it is necessary to parse and analyze large amounts of source code. Furthermore, none of the existing program analysis tools worked very well for our particular problems. While many of the existing analysis tools are successful in a number of ways, they are typically difficult to integrate or extend into new research and products. The existing tools are typically: tightly coupled with other tools, language dependent, or embody a methodology orthogonal to the specific problem. Additionally, these tools are given little support by the original developers and/or require specific (older) OS versions, platforms, and libraries. These inherent problems have also been described by others [Favre'03, Tilley'03]. Oftentimes using and modifying an existing tool is as difficult as building your own from scratch.

Many of our colleagues run into this same problem. A number of these researchers expressed the simple need for an easy to use C++ parser that allows them access to the abstract syntax tree. Of course, one can hack g++ and recent versions allow some access to this information but it is still very difficult to integrate into other tools. Also, the intermediate output from a compiler is only part of the information we (and others) need. A large amount of information is lost in the compilation process and this is often vital to support such things as program understanding tasks.

To address our own problems we felt that building a tool (or set of tools) that is easy to integrate and easy to use was necessary to the long term goals of our research. Also, such a tool should be easily shared with others working on similar problems. Our need to extract syntactic information from C++ came to a head in early 2001. While we previously used quick-and-dirty solutions, these types of methods no longer supported our problem. So, we decided to build our own generic tool(s). At the same time, XML technologies had been around for a good duration and were quite mature. We'd been thinking about using XML to represent source code since the late 90's and some research in this direction had been conducted [Badros'00, Cox'99].

These days, any-type of new data storage or manipulation is using, is considering, or being compared to XML. This is due to the wide variety and availability of XML technologies including formats, tools, and standards. In the case of access to XML data there are pull and push parsers and XML transformation languages (e.g., XSLT, STX, and TextReader). Choosing an XML representation allows the use of many (if not all) of these powerful tools.

Many developers and researchers are using or are planning to use XML technologies for the applications they are constructing. Although some of these people may not be completely familiar with the intricacies of XML Schema and XQuery they do understand the basics and can quickly develop applications using the ease to use technology.

Our approach is not so much a tool as it is a representation. We designed a XML format, *srcML* [Collard'02, Maletic'02], to mark up source code (C/C++/Java) with explicit syntactic information. In essence the abstract syntax of the program is directly imbedded into the source code. So there is no need to do

parsing after the source has been translated into srcML. Importantly, the representation does not alter the original source code in any inherent manner. It preserves all the formatting, comments, macros, etc. This is vital to support our own research and quite different from most other approaches along with being a stated requirement for many software engineering tasks [Van De Vanter'02].

With source code marked up in srcML we argued that all of the existing XML tools and technologies can be leveraged to construct simple tools in an opportunistic manner to solve a particular problem. We developed a C++ to srcML translator and demonstrated our approach by addressing the problem of C++ fact extraction in the context of a lightweight XML approach [Collard'03]. Using our approach has proven to be quite successful with regards to flexibility, integration with other tools, platform independence, easy of use, and adoption of our tools.

In the remained of this paper we will briefly describe srcML and how we utilize XML tools to solve various problems. We also discuss the requirement of our approach within the context of adoption as we believe many of our decision directly support this nonfunctional requirement. How we support srcML currently and plan to in the future is described along with the requirement of using our approach. We conclude with a summary of issues that support the construction of adoption centric software.

2. XML and Source Code

A number of options have been investigated for representing source code information (e.g., AST or ASG) in a XML data format namely, GXL [Holt'00], CppML [Mammas'00], ATerms [van den Brand'98], GCC-XML, and Harmonia [Boshernitsan'00]. In these formats the abstract syntax tree (AST) (actually a graph) of the source code, as output from a compiler intended for code generation, is stored in a data XML data format. These XML data views of source code, since they are based on the AST, are a "heavyweight" format that requires complete parsing of the original document and generation of the complete AST. The work most closely related to srcML is Badros' work on JavaML [Badros'00], which is an XML application that provides an alternative representation of Java source code.

However many of these approaches do not take full advantage of XML technologies. They may provide XML output or input, but not use it internally, take a data view of XML at the expense of a document view, or provide an XML API for accessing the data but not allow for the full range of XML application and they may not be designed for the full variety of XML processing (i.e., pull-parsing, push-parsing etc.). The use and development of XML tools is an active area of research and it is difficult to predict what tools or languages a developer may choose to use with an XML format now or in the future.

2.1. srcML

srcML (SouRce Code Markup Language) [Collard'03, Collard'02, Maletic'02] is an XML application that supports both document and data views of source code. The format adds structural information to raw source code files. The document view of source code is supported by the preservation of all lexical information including comments, white space, preprocessor directives, etc. from the original source code file. This permits transformation equality between the representation in srcML and the related source code document.

A lightweight data view of source code is supported by the addition of XML elements to represent syntactic structures such as functions, classes, statements, and entire expressions. Other structural information including macros, templates, and compiler directives (e.g., #include), are also represented. The data view stops at the expression level with only function calls and identifier names marked inside of expressions thus allowing reasonable srcML file sizes. The srcML for the simple program below is given in Figure 2.

```
#include <iostream>

// A function
void
f(int x)
{
  std::cout << x + 10;
}
```

In this example we see that all of the original text is present, including the preprocessor directive `include` and all original comments. Some of the original text that

```
<unit xmlns="http://www.sdml.info/srcML/src" xmlns:cpp="http://www.sdml.info/srcML/cpp">
<cpp:include>#<cpp:directive>include</cpp:directive><cpp:file>&lt;iostream&gt;</cpp:file></cpp:include>

<comment type="line">// A function
</comment><function><type>void</type>
<name>f</name><formal_params>(<param><type>int</type> <name>x</name></param>)</formal_params>
<block>{
  <expr_stmt><expr><name>std::cout</name> &lt;&lt; <name>x</name> + 10</expr>;</expr_stmt>
}</block></function>
</unit>
```

Figure 1. Example of srcML. Notice that preprocessor directives are marked up and not expanded. The format preserves all textual context entered by the developer.

are meta-characters in XML, e.g., '&', have been encoded but all of the other text is preserved. The documentary structure of the original text including spacing and lines are also preserved. The inserted XML tags allow for the addressing and location of textual elements according to their location in the XML document.

The data view allows for a search-able and query-able representation. This can be mixed with a document view to permit multiple levels of abstraction (or views), and allows a data view of the document without losing any of the document information. The reverse is also allowed with document information, e.g., white space, comments, etc., used in the data view for searches or queries.

Once the document is in srcML locations in the srcML document (and corresponding locations in the textual source code document) can be referred to using the XML addressing language XPath. For example, to refer to the second if statement inside of a function named foo we can use the following XPath: //function[name="foo"]/ block/if[2]. Using XPath we can address the document using a variety of paths to refer to locations. XPath can also be used to represent groups of elements, such as all functions.

The capability to use XPath addresses is built into most XML tools and is used extensively in XML transformation languages such as XSLT. The XPath standard forms the base of the XML query language XQuery.

Unlike the physical representation of an address that a line number references, XPath addresses describe one of many possible paths to a location. The XML elements serve as reference points along the path. This makes XPath addresses much more resilient to changes in other parts of the document, unless they change the nested XML elements.

2.2. Adoption of srcML

In the design and construction of srcML there are a number of factors that have an effect on the adoption of this type of approach. In short, selecting a light weight philosophy is behind many of these issues. We discuss each issue as a postmortem in the context of adoption.

Choice of document view over data view. XML applications (e.g., XML formats) and processing falls into the categories of a document view (e.g., DocType, etc.) or of a data view (e.g., SOAP). These two views influence the layout, semantics, and tools for the format (e.g., from a document view white space is of importance, from a data view white space is of no importance). Source code, especially in an unprocessed state, is closer to a document than to data. That is, we program by writing documents, not by specifying parse trees. When required to make a decision between the two views, the document view is the clearest choice in this case. This does not lead to any loss since a data view can simultaneously be supported with careful handling during processing and analysis.

Preservation of the original data (i.e., source code). Allowing all of the original data to remain in any kind of transformation process is an accepted data manipulation design principle. This is true even if the data is not needed. In srcML the preservation of all original text allows for the restoration of the complete original document. The format does not try to reorganize or change the text. It only augments the text with markup that extends its capabilities while still allowing the added information to be easily extracted.

Markup only what is of interest. In srcML the markup stops at the expression level (i.e., expressions and the identifiers contained in the expressions are given markup). The full AST of the expression is not given markup because this does not meet the requirements of the uses that we saw. While marking full expressions down to the operator and parentheses level may be useful for expression rewriting and other compiler-oriented tasks, it is not useful for the identified tasks.

Tag Names based on programmers view. Programmers know the basic syntax of the language that they are programming. They also know the syntactic name of many of the program structure, i.e., block, function, type, etc. They typically do not know the exact distinction between the finer points of syntax naming. For example, in the function declaration: const int foo(); most programmers most likely will identify the return type of the function as const int. In srcML the type const int is marked using the tag <type>. Technically however, the type is int and const is a type specifier. Most programmers only care about this distinction when reading syntax diagrams.

Minimization of meta-data. There are very few attributes used in srcML. The markup in the source code is to provide navigation and access to the content. Information that may be derived from other parts of the program, e.g., the type of a variable used in an expression, are derived versus stored as attributes.

Easy conversion to and from the original format. Source code text is extracted from directly from srcML by removing markup and some output un-escaping. This is done by a variety of XML tools or even by simple Perl or Python programs.

Lightweight schema. There is a tendency to produce a source code format that is very stringent, i.e., any documents produced in the representation can only represent compilable C/C++ programs. Our philosophy is that this is unnecessary since there are compilers to test if a program can compile. We see this as a hindrance since the source code may be in a state that can not be compiled (e.g., it is under maintenance or during

refactoring). This implies that we can do analysis on incomplete programs, programs with missing libraries, and source code that is under construction.

Format efficiency. One of the criticisms of XML formats is their size. Data represented in an XML format can balloon up as levels of markup and attributes are used. This may cause an increase in size of hundreds of times over the original document and is especially problematic to DOM approaches (including XSLT) which store the entire tree in memory before processing is started. In practice, a srcML document is on average 5 times larger than the original source code document. Full AST markup in XML can result in hundreds of times increase of file size [Power'02].

3. srcML Translation

Translating source code into srcML allows the source code to be integrated into an XML infrastructure. The translation process directly influences how completely the XML infrastructure can be utilized and what applications can take advantage of the format. We now describe the C/C++ to srcML translator and the influences with regards to adoption.

3.1. srcML Translator

The srcML translator takes as input C/C++ source code text and inserts XML around the syntactic structures to produce srcML. The translator has additional requirements over that of a traditional parser (e.g., in a compiler) in preservation of source code text and ability to work with code fragments and incomplete code. Design decisions were made that meets these requirements to support all of the features of the srcML representation. In this section the important design considerations are briefly discussed.

Existing compiler-centric parsers have difficulties in the preservation of the original source code text especially with regard to white space, comments, and preprocessor directives. In addition they are not designed to handle incomplete code. Regular expression pattern matching approaches do not have these problems, but have difficulties in the context of what they are matching.

Typical parsers take a LR(k) or bottom-up approach (e.g., parsers generated by yacc). They start with parsing lower level components and use multiple production rules for reduction. Lower level syntactic elements are parsed and identified before higher level structural elements, e.g., contents of a block are parsed before the block structure itself.

A top-down parser generated by the compiler generator ANTLR was used to identify and markup the elements in the source code as per the natural structural order of the components. Minimal parsing was used to identify top level structural elements before constituent elements (e.g., identify that a function definition had started without having to parse all of its contents).

The translator employs a selective parsing approach based on the concept of island grammars [Moonen'01]. The parsing of a language construct such as function definition occurs in multiple passes with the identification and markup of the start of the function definition done in the first pass.

The C and C++ languages have a non-CFG (Context Free Grammar). Traditional compilers construct a symbol table which can be used to resolve non-CFG ambiguities. However a complete symbol table is not possible with code fragments and incomplete code. The translator handles this issue by considering a CFG view of the non-CFG C/C++ grammar.

Most parsing and markup methods follow a batch sequential architecture: parse the source code, generate the entire AST, insert tags at appropriate nodes, and output the tree with the markup. The top-down parsing of ANTLR was extended to stream parsing where XML tokens were inserted into the stream of text tokens as soon as a markup element is identified. XML tags were only wrapped around syntactic elements of (high) interest in a language construct (e.g., in case of an expression statement identifier names are marked up while arithmetic operators are left unmarked).

The translator is the first stage from C++ source code to srcML representation. The srcML output is designed to be refined with external parsing stages based on processing of associated source code files, user defined heuristics, and user knowledge.

3.2. Adoption of the Translator

In addition to the common requirements of an application such as accuracy, reliability, and speed, the srcML translator had the additional requirement that it be able to be fully integrated with XML technologies.

Responsiveness. The decision to use event-parsing allows for output as soon as a statement is detected. Since output is immediately available the translator can be integrated into a stream XML processing (i.e., SAX, STX, TextReader) allowing for the efficiencies of memory that these approaches allow. Although not as important for tree XML processing (i.e., DOM, XSLT) it does allow for simultaneous translation and tree building.

Flexibility. It is important to support all possible forms of usage of any tool. Programmers still often use CLI tools (e.g., grep) because they are fast, portable, and easy to use once learned. They also are easily scriptable leading to low-level productivity tools. CLI tools can also be used in a GUI.

Scalability. The event-driven translation approach allows for its use on large source code files and large

collections of source code files. The translation is nearly linear.

Extensibility. The srcML translator's CFG view of the source code and output in XML provides a base for further source code processing. This processing may more accurately markup the source code based on external information (i.e., a symbol table), or transform it for another usage entirely.

Portability. The srcML translator is a CLI program that can be used on both MS Windows and Linux.

Robustness. The translator must generate a well formed srcML document no matter what the state of the input source code. This is related to the lightweight view of schema conformance.

Once the translator met this list of requirements it was able to be used with the complete range of XML technologies. In the next section we will describe some of the uses of XML with srcML for program analysis tasks.

4. Applications using srcML

As with the many applications where XML is utilized we see a wide range of applications for srcML within the context of development, analysis and transformation of source code. The following sections cover the current applications that leverage the XML infrastructure technology with our srcML representation and translator. The first section discusses the direct use of the srcML representation and the next section covers applications that extend the srcML representation.

4.1. Directly Leveraging srcML

XML technologies have been combined directly with the srcML representation to provide applications ranging from viewing, searching, and editing source code to source code transformation. XML began as a document format with support for style-sheets formats such as CSS and XSL-FO, therefore source code viewers are a natural application using srcML. CSS style-sheets have been used to pretty print source code, hide or emphasize particular program elements, and perform simple abstract visualizations. Web browsers that support XML (e.g., Internet Explorer, Mozilla/Netscape) can be directly used as source code viewers.

Moving beyond viewing is of course editing using srcML. The user edits the source code normally, i.e., by editing text and in the background the srcML translator generates the corresponding srcML. The srcML can be used to control the view of the source code using style-sheets. Editing has additional requirements over viewing. In editing we cannot assume that the source code is in a compilable, complete state and so srcML must be able to represent source code that is not in a compilable, complete state. The translator must also accept this realistic view of source code, while at the same time providing the responsiveness that we expect in an interactive application. An editor that utilizes these features of srcML is currently in development.

The srcML format is being used to extend the capabilities of source code search tools. Current search tools are based on regular expressions in which it is difficult to define the proper context of the matching in terms of the syntactic structure, e.g., an identifier name in a comment.

The srcML tags provide XML reference points for addressing locations in source code documents. Searches are expressed using the XML addressing language XPath. Many XML tools evaluate XPath expression, e.g., the command line utility *xpath*. Full queries can also be performed on source code by the application of XPath. This has already been used for our C++ Fact Extractor that combined the srcML translator, a command line XPath tool (*xpath*), and XPath expressions. If a full query language is needed XQuery (XPath is a subset of XQuery) can be used, or any other XML query language that is developed.

The preservation of the original source code text in the srcML allows source code transformations to be performed using XML tools. Source code is converted to srcML, transformed using XML transformation languages and API's, and converted back to source code. An identity XML transformation of srcML is an identity transformation of the original source code text preserving the programmer's view of the source code.

Source code transformations with srcML can use any XML transformation language or tool including the XML transformation language XSLT, or the use of an API in a more traditional programming language, e.g., Python using a DOM API. Because of the responsiveness of the translator memory efficient pull-parsing XML API's such as SAX and languages such as STX can be used.

4.2. Extensions of the srcML Representation

A specific, and exciting, transformation application is to support aspect weaving. Aspect Oriented Programming (AOP) is a programming paradigm that addresses the problem of advanced separation of concerns. Aspects, written separately from the base code, are woven into the base code based at specific locations, e.g., before a function return. The XWeaver Project at ETH (http://control.ee.ethz.ch/~ceg/XWeaver) uses srcML to define aspects in the format AspectX. srcML is used to represent the code to insert (aspect) and the location in the code were the aspect is to be inserted. The aspects are woven into the source code using a XML transformation for non-intrusive source code transformations.

The srcML format is being used for differencing applications (results of this work are submitted to ICSM'04). Current differencing tools and approaches are either efficient with a low-level of information (e.g., the utility *diff*), or inefficient with a higher-level of information (e.g., semantic differencing). The srcML format has been extended to support Meta-Differencing, i.e., the extraction of higher level information from the differences in source code documents. This information is stored in srcDiff, a multi-version extension of the srcML format.

The srcDiff format extends the querying capabilities of srcML for queries involving both the contents of the source code and versioning information. The srcDiff document is constructed by weaving the two versions of the srcML documents together with the utility *diff* indicating where differences occur.

The srcML format can also be used for representing higher-level structures in the source code as with source models. Source models provide an abstraction of the source code that focuses on the concept of interest hiding unnecessary details. They are often constructed after further analysis of the source code and may include information that is not contiguous or singular in the source code.

5. Conclusions

Using XML in the context of adoption presents a number of advantages including support for multiple query languages, support of complex transformations, support for a document format as well as a data format, broad based usage and acceptance of standards, and a large interest from open source community. The general academic and industry communities have embraced XML and its associated tools.

In our experience spending additional effort to construct an underlying XML format that takes advantage of all the tools is critical to a successful application. Additionally, we focused not on solving a particular task but instead we built an infrastructure to address a general set of problems. This allows the user to opportunistically use XML tools to address their particular problem. We feel adoptable software should address general problems and allow for specialization to particular tasks.

6. References

[Badros'00] Badros, G. J., (2000), "JavaML: A Markup Language for Java Source Code", in Proceedings of 9th International World Wide Web Conference (WWW9), Asterdam, The Netherlands, May 13-15.

[Boshernitsan'00] Boshernitsan, M. and Graham, S. L., (2000), "Designing an XML-Based Exchange Format for Harmonia", in Proceedings of Seventh Working Conference on Reverse Engineering (WCRE'00), Australia, Nov. 23-25, pp. 287-289.

[Collard'03] Collard, M. L., Kagdi, H. H., and Maletic, J. I., (2003), "An XML-Based Lightweight C++ Fact Extractor", in Proceedings of 11th IEEE International Workshop on Program Comprehension (IWPC'03), Portland, May 10-11, pp. 134-143.

[Collard'02] Collard, M. L., Maletic, J. I., and Marcus, A., (2002), "Supporting Document and Data Views of Source Code", in Proceedings of ACM Symposium on Document Engineering (DocEng'02), McLean VA, Nov. 8-9, pp. 34-41.

[Cox'99] Cox, A., Clarke, C., and Sim, S., (1999), "A Model Independent Source Code Repository", in Proceedings of Proceedings of the IBM Center for Advanced Studies Conference (CASCON'99), November 8-11, pp. 381-390.

[Favre'03] Favre, J.-M., Estublier, J., and Sanlaville, A., (2003), "Tool Adoption Issues in a Very Large Software Company", in Proceedings of 3rd International Workshop on Adoption-Centric Software Engineering (ACSE'03), Portland, Oregon, USA, May 9, 2003, pp. 81-89.

[Holt'00] Holt, R. C., Winter, A., and Schürr, A., (2000), "GXL: Toward a Standard Exchange Format", in Proceedings of 7th Working Conference on Reverse Engineering (WCRE '00), Brisbane, Australia, November, 23 - 25, pp. 162-171.

[Maletic'02] Maletic, J. I., Collard, M. L., and Marcus, A., (2002), "Source Code Files as Structured Documents", in Proceedings of 10th IEEE International Workshop on Program Comprehension (IWPC'02), Paris, June 27-29, pp. 289-292.

[Mammas'00] Mammas, E. and Kontogiannis, C., (2000), "Towards Portable Source Code Representations using XML", in Proceedings of 7th Working Conference on Reverse Engineering (WCRE '00), Brisbane, Queensland, Australia, November, 23 - 25, pp. 172-182.

[Moonen'01] Moonen, L., (2001), "Generating Robust Parsers using Island Grammars", in Proceedings of 8th IEEE Working Conference on Reverse Engineering (WCRE'01), Suttgart, Germany, October 2-5, pp. 13-24.

[Power'02] Power, J. F. and Malloy, B. A., (2002), "Program Annotation in XML: a Parse-tree Based Approach", in Proceedings of 9th Working Conference on Reverse Engineering, Richmond, Virginia, October 2002, pp. 190-198.

[Tilley'03] Tilley, S. R., Huang, S., and Payne, T., (2003), "On the Challenges of Adopting ROTS Software", in Proceedings of 3rd International Workshop on Adoption-Centric Software Engineering (ACSE'03), Portland, OR, May 9, 2003, pp. 3-6.

[Van De Vanter'02] Van De Vanter, M. L., (2002), "The Documentary Structure of Source Code", Information and Software Technology, vol. 44, no. 13, October 1, pp. 767-782.

[van den Brand'98] van den Brand, M., Sellink, A., and Verhoef, C., (1998), "Current Parsing Techniques in Software Renovation Considered Harmful", in Proceedings of 6th International Workshop on Program Comprehension (IWPC'98), Ischia, Italy, June 24-26, pp. 108 - 117.