European Union
European Social Fund

OPERATIONAL PROGRAMME
EDUCATION AND LIFELONG LEARNING
investing in knowledge society
MINISTRY OF EDUCATION & RELIGIOUS AFFAIRS
MANAGING AUTHORITY
Co-financed by Greece and the European Union

NSRF 2007-2013
programme for development
EUROPEAN SOCIAL FUND

SPRINT SMEs PROJECT

SPRINT SMEs Project: Research in Software PRocess ImprovemeNT Methodologies for Greek Small & Medium sized Software Development EnterpriseS

Work Package 2 (WP2): Requirements Analysis for SPIRE (Software Process Improvement in Requirements Engineering) Projects

Deliverable 2.1 (D2.1): Requirements Analysis for Software Process Assessment/Improvement Frameworks

Authors:

- Vassilis C. Gerogiannis (Project Leader, Project Scientific Coordinator), Associate Professor, Department of Project Management, Technological Education Institute of Thessaly, Greece
- George Kakarontzas, PhD, Application Professor, Computer Science and Telecommunications Department, Technological Education Institute of Thessaly, Greece
- Dimitris Tselios, MSc, PhD Candidate, Application Professor, Department of Project Management, Technological Education Institute of Thessaly, Greece
- Ioannis Stamelos (WP2 Leader), PhD, Associate Professor, Department of Informatics, Aristotle University of Thessaloniki, Greece
- Stamatia Bibi, PhD, Research Associate, Department of Informatics, Aristotle University of Thessaloniki, Greece

July, 2013

Research in SPRINT SMEs project is implemented through the Operational Program "Education and Lifelong Learning" and is co-financed by the European Union (European Social Fund) and Greek national funds in the context of the R&D program "ARCHIMEDES III"

TABLE OF CONTENTS

Part I: State of the Art in Requirements Engineering

1. Background on Requirements Engineering

What is Requirements Engineering?

The success of a software project is highly dependent on the degree that the project outcome (i.e., the developed software system/service) satisfies the needs of its users and its environment and software requirements encompass these needs [1]. Requirements Engineering (RE)[1] is the process that emphasizes on determining/analyzing requirements and involves [2, 3]:

i) discovering the purpose for which a system has to be developed,

ii) identifying system stakeholders and their needs,

iii) understanding the environmental context in which the system will be used,

iv) negotiating,

v) prioritizing,

vi) modeling,

vii) analyzing and

viii) documenting requirements in a form that is amenable to communication, implementation and testing,

ix) validating that the documented requirements are in line with the identified stakeholders' needs,

---

[1] In this deliverable, we prefer to use the term "Requirements Engineering", instead of the term "Requirements Analysis", since the former more generallly refers to all processes relevant with software requirements.

x) verifying that requirements are adherent to quality attributes, and

xi) managing requirements as they evolve.

In the first part of the present deliverable, a state of the art survey is presented in the area of RE, that conistutes one of the target research orientations in the context of the SPRINT SMEs project (Research in Software PRocess ImprovemeNT Methodologies for Greek Small & Medium sized Software Development Enterprises - SPRINT SMEs).

In particular, Part I of the deliverable presents the problems solved through the RE process, current research and industry status in the field, a classification of RE methodologies and tools as well as anticipated future research trends in the area.

In Part II of the deliverable, we concentrate our emphasis on the current state of practice in the RE methods of components adaptation and reuse, which are also of particular interest in the context of the SPRINT SMEs project. The reason for this interest is justified since components adaptation/reuse methods can offer means for Small and Medium Sized Enterprises (SMEs) to reduce development expenses and achieve benefits that would be otherwise impossible to achieve.

What problems are solved through the Requirements Engineering process?

The "requirements mess" problem [4] is a continuous challenge in Software Engineering (SE) research and practice. Although there are various perceptions on defining the success/failure of a software project [5, 6], most academic and industrial research efforts demonstrate (see, for example, [7, 8, 9, 10]) that among the leading sources of a software system success or failure there are factors related to inadequate diligence at the RE process.

Examples of these sources are:

i) misunderstanding/lack of user input & involvement,

ii) incomplete/rapidly changing requirements, and

iii) extremely complex relationships between software systems of today and their technological/organizational environments.

By adopting a straightforward classification scheme for the several tasks comprising a RE process [3, 11], we can realize some conventional problems addressed:

i) discovery/elicitation activities enable understanding/identification of organizational/stakeholder objectives & needs,

ii) analysis/specification tasks transform the elicited requirements in terms of a representational notation by using either informal or formal models,

iii) validation tries to ensure that requirements reflect stakeholders' needs, while verification examines the degree that requirements conform to quality criteria, such as consistency, feasibility, traceability and lack of ambiguity, and

iv) requirements management focuses on identifying changes over time and across different software product families, establishing and documenting traceability links, determining maturity/stability of elicited requirements, performing cost-benefit analysis for requirements' changes, enhancing maintainability, administering and prioritizing large number of requirements, which may be distributed geographically, functionally and organizationally.

What is the status of research in RE?

RE engineering research/practice is "active" for more than 30 years, since practitioners in early 1970's noticed the benefits of adopting a systematic undertaking of requirements in a software development project. Despite the significant progress in the relevant body of research (improved software tools, modeling methods and process technologies for RE), there are always difficulties in applying an efficient/effective RE process.

The economics of RE is always a crucial success factor, since the cost of resolving requirements errors and handling requirements volatility is growing exceedingly, as software life-cycle evolves [12]. The main source for RE difficulties arises from the distinctive "nature" of RE compared to other SE processes [3]. RE focuses on the (large) *problem space area*, by defining the problem that is to be solved by a software system, while other SE processes emphasize on the (more limited) *solution space area*, by defining/testing the behavior of a software system.

RE starts with analyzing an ambiguous problem description and defining the system environmental conditions, and must, finally, result in a concrete technical solution that is amenable to implementation. However, the conditions in the environment of a software system can be various, complex and unconstrained and thus, the complexity of requirements tasks is getting higher. The intrinsic limitation of human information processing and obstacles in interaction/mutual understanding between designers and multiple stakeholders also make RE an inherently difficult process.

RE should emphasize on reconciling the relationships between non-technical staff (customers/users) and technical staff, gaining involvement from all parts and providing results (requirements artifacts) which are understandable and useful to all.

All the above limitations create a constant interest in continuous re-think, re-align and re-evaluate RE practices to cope with practical challenges and emergent demands.

What is the current status of RE adoption? How much mature is RE?

A number of RE methodologies, techniques and tools have gained a wide adoption in practice [13], e.g.,

i) elicitation techniques like focus groups, stakeholder interviews and exploratory prototyping,

ii) business process, system modeling notations, structured requirements documentation, object-oriented modeling by using UML (especially use case diagrams) and supporting CASE tools, risk-oriented RE,

iii) traceability analysis,

iv) RE process improvement (CMMI),

v) service-oriented architectures,

vi) outsourcing of development etc.

Yet, the field cannot be considered as mature, since new challenges appear as new needs from software are emerging. The interest in RE advances is growing up, mainly due to the evolving complexity, heterogeneity and uncertainty of the technological/organizational environments where the software is developed or utilized [14]. Requirements of today software systems are shaped by the rapid change in technological capabilities, new application demands in rapidly evolving environments and the need of developing systems for maximum value creation.

The success of RE process (followed in a RE project) is also more dependent on complex organizational issues such as the politics of resource allocation and the legitimacy of decision making [15]. Principles of information hiding and data abstraction are no longer enough to handle the “requirements mess”. New principles, multidisciplinary research strategies and key issues are required to be investigated and adopted for handling the new shift of the field’s focus, that is, the transition from engineering isolated systems and components to generating and adapting IT-supported ecologies of systems [14].

In a strict sense, there are no competing alternatives to RE. Even in iterative projects which do not emphasize on formal documentation (e.g., in agile software development), solutions evolve through feature-driven development and communication/collaboration between self-organizing/cross-functional teams [16]. However, the groundwork of RE is highly “competitive”. RE is a multidisciplinary field that merges many sciences to suggest the theoretical foundation and practical methods for eliciting, modeling, analyzing and managing requirements [1]. The groundwork of RE draws not only upon software/systems engineering, but also upon theoretical/practical computer science, logic, cognitive psychology, anthropology, sociology, linguistics and philosophy (epistemology, phenomenology, ontology).

2. State of the Art in Requirements Engineering Research

Recent state-of-the-art reviews in RE are performed by classifying the activities of a RE process into phases, which are executed rather iteratively and progressively [3, 11]:

i) elicitation/discovery,

ii) modeling/analysis/specification,

iii) validation/verification, and

iv) requirements management.

Table 1 (adopted mainly from [3] and enhanced with findings from [1] and [11]), summarizes corresponding methodologies and techniques that have been proposed in the relevant literature.

Table 1: Requirements Engineering Methodologies & Techniques

RE Phase/Activities	**Methodologies / Techniques**
Discovery & Elicitation	• Techniques for stakeholders identification, • Analogical techniques (Metaphors, Personas), • Techniques for inventing requirements (questionnaires, surveys, brainstorming, interviews, focus groups, consensus-building workshops, e.g., RAD/JAD workshops), • Direct observation, • Cognitive techniques (e.g., Protocol analysis, Laddering, Card sorting, Repertory grids), • Contextual Techniques (e.g., Ethnography, Ethnomethodology, Conversation analysis), • Exploratory prototyping, • Models animation & simulation, • Invariants generation, • Storyboards, • Exploratory models (Use cases & Scenarios, e.g., CREWS), • Enterprise models, • Policy & Goal models, • Goal-based methods (e.g., KAOS and i*), • Agents, • Elicitation of security requirements (Anti-models, Trust assumptions, Threat models), • Elicitation of non-functional requirements
Modeling	• Formal models, • Natural language representations, • RE Reference models, • Viewpoints, • Patterns, • Modeling facilitators, • Formalization heuristics, • Models merging/synthesis/composition, • Enterprise modeling (business goal, rule & workflow models), • Behavioral & functional models, • Structured analysis & design,

	• Object-oriented models, • Data models (ERD, Class/Object diagrams), • Domain models & descriptions, • Property languages, • Notation semantics, • Goal-based models, • Agent models, • Anti-models, • Threat models, • Non-functional requirements models
Analysis	• Negotiation, • Requirements analysis with respect to Components Off The Self (COTS), • **Components adaptation and reuse**[2], • Conflict management & analysis, • Linguistic analysis, • Ontologies, • Checklists, • Inspections, • Obstacle analysis, • Risk management, • Impact analysis, • Causal order analysis, • Requirements selection techniques (from multiple viewpoints), • Requirements prioritization, • Variability analysis, • Automated reasoning (e.g., analogical & case-based reasoning), • Consistency checking
Validation & Verification	• Model based Validation & Verification (V&V), • Prototyping, • Disagreement & negotiation management, • Stakeholder voting techniques, • Prototyping,

[2] The methods of components adaptation and reuse are of particular interest in the context of the SPRINT SMEs project, since they offer means for Small and Medium Sized Enterprises (SMEs) to reduce development expenses and achieve benefits that would be otherwise impossible to achieve. The current state of practice in the area of component adaptation/reuse is presented in PART II of the current deliverable.

	• Mock-ups, • Validation of use-case personas, • Validation of time estimates, • System walkthroughs, • Simulation, • Animation, • Invariants generation, • Model checking, • Consistency checking, • Satisfiability checking, • Automated reasoning
Management	• Scenario management, • Feature management, • Configuration management, • Version control, • Viewpoints, • Cost-benefit analysis, • Financially informed feature management (e.g., Incremental Funding Method), • Globally distributed RE, • Traceability management, • Stability analysis, • Variability modeling, • Requirements management for COTS-based systems, • Requirements management for software product families

There is also an extensive amount of valuable research results in RE software process improvement (SPI) methods which can be classified into two broad categories: inductive (or problem-based) [17] and prescriptive (or model-based) [18, 19]. The former assess process improvement upon the strengths/weaknesses of existing processes and consider as knowledge sources the stakeholders' perceptions, while the latter, and more dominant in the SPI area, center process improvement upon best practices within industry.

3. Current State of Practice in Requirements Engineering

The current practice in the field of RE is rather limited, compared to the large relevant research work in the area. Field surveys and empirical studies observe (e.g. [11]) that there is a persistent divergence between research and practice, as practitioners "slowly" adopt requirements methodologies/techniques proposed by researchers often "ignore" practices and actual needs of requirement engineers/designers.

In particular, a recent field survey, involving 39 RE professionals from Fortune 500 companies [11], demonstrates the wide acceptance of a rather limited subset of requirements practices, such as:

i) risk mitigation and control,

ii) use of semi-formal modelling techniques (e.g. use of UML, particularly UML use cases, and supporting CASE tools),

iii) requirements elicitation, through focus group discussions and cross-disciplinary stakeholder teams, and

iv) requirements validation, through prototyping, mock-ups and system walkthroughs.

Traceability analysis and software process improvement (e.g., CMMI [18], ISO/IEC 15504 – SPICE [19] or "lightweight" process improvement models [17]), although not widely accepted, constitute directions of interest for many companies, especially for SW SMEs. Effective balancing the challenges of specificity, comparability and accuracy in a process assessment/improvement project, by considering also resources' constraints, is another practical problem, especially for SW SMEs interested in investing resources at an SPI project [20].

Furthermore, Service-Oriented-Architectures (SOA) and outsourcing of development are influencing requirements practices in many companies, while issues like the systematic treatment of non-functional requirements, use of formal methods and methodical handling of conflicts among stakeholders, are not among the commonly applied practices.

4. Conclusions of Part I: Anticipated Research Trends & Evolution of RE

Designing and implementing an effective/efficient RE processes is still a very challenging problem. Contemporary large-scale systems are actually Information Technology (IT) supported ecosystems with highly dynamic intra-dependencies among their subsystems and inter-dependencies with their (technical/social) unstable environment. Therefore, current and close future needs include the support of tighter integration of software with its operational/organizational environment, greater self-efficiency of software to adapt in environmental changes as well as transparency/seamless user experience across different applications.

The economics of RE and software engineering, in general, has changed. “Speed” of requirements (i.e., volatility of requirements), time-to-market and low cost demands, end-user development, increasing globalisation of software through outsourcing/off-shoring development are driving forces which create new challenges for RE. RE is also required to address requirements of software applications developed by considering different design perspectives, such as industrial design (pervasive applications), media design (e-commerce and media software), interaction design (applications with multi modal interfaces) and business environment design (open business platforms).

European R&D in the area of RE will certainly improve the success rate of ICT projects and the competiveness of European SW SMEs, in an arena that is characterized by emerging demands to capitalize on global resource pools, decrease development costs, develop innovative, widely accepted and more usable software systems which reflect changing users’ needs. Many new challenges in RE have arisen in the recent years, while the transfer of new practice problems to research has been less successful. Therefore, the European RE research & practice community needs to be proactive in identifying RE research problems and solutions which will leverage successful and practical results.

By taking into account all the above emerging critical needs, some representative key research areas which are likely to be of interest in the future are the following [3, 11, 13]:

Requirements modeling, abstraction and analysis techniques for large-scale and ultra-large-scale systems

Systems scaling is no longer related only with system size parameters, but also with complexity, heterogeneity, uncertainty, environment variability, number of interacting devices and interfaces, number of distributed decision/computation making nodes, etc. In the close future we will require more systematic RE methodologies and rigorous techniques to cope with the complexity of large-scale systems (e.g., next generation command & control systems, future intelligent transportation systems, systems managing power grids, systems controlling telecom infrastructures, integrated health-care systems, systems which are integrated with legacy infrastructures etc.).

Potential problems are the diversity of requirements and stakeholders' concerns, the need to integrate multiple disciplines (sensors communication, scientific computation, intelligent scheduling & control), new abstraction/refinement paradigms and conflict management techniques for large-scale systems.

Security RE for pervasive & mobile systems and flexible tolerance in critical applications

Since systems become more mobile, pervasive and ubiquitous, they are more potential to be targets of exceedingly emerging security threats/attacks. Thus, additional research efforts are required towards low-level security RE (e.g. identify/avoid vulnerabilities, protect information, specify counter measures to handle attacks and recover from attacks), but also towards structuring and reasoning on high-level security requirements (e.g., security requirements for organizational structures and modeling/analyzing security policies) and examining the intertwining between low-level and high-level security.

The complexity of future critical systems (e.g., software systems for transportation vehicles, medical applications and command & control systems) increases the need to focus more on tolerance requirements of these systems in order to present more adaptable behavior to dynamically changed environments. Issues of specifying levels of acceptable behavior in (fault) tolerant systems, relaxed correctness requirements as well as diagnostic and recovery mechanisms are expected to be further investigated.

RE techniques for highly integrated systems

The current trend of developing integrating applications rather than developing new ones is expected to be continued. There is a variety of new generation highly integrated systems in which computing is tightly interleaved with environmental monitoring and control. Examples are intelligent transportation systems, intelligent manufacturing control systems, energy consumption optimizers and smart wearable computing systems.

Systems integration, particularly in business domains, is driven by user considerations, while there are additional demands for providing seamless user experience and transparency across different platforms. Therefore, research in requirements analysis for highly integrated systems is anticipated to focus more on analyzing the interdependencies among physical environment, human behavior, interfaces and software system.

RE techniques for self-managing systems

Self-management capabilities are essential for software systems which are aware of their context and continuously react and adapt to contextual changes. Examples are mobile devices which provide services according to the user location, control systems able to recover dynamically from failures and intrusion detection systems which handle security breaches at run time. For this modern class of systems, the current research work on specifying and verifying their adaptive behaviour (e.g., [21]) is expected to gain a more growing interest.

Another promising direction of research is investigating theories and models from other disciplines (e.g., biology) in order to define metaphors for discovering and analysing requirements for future adaptive, self-managing computer systems [22].

Effective/efficient techniques for globally distributed RE

Another current trend in software development is the increased distribution of requirements processes across functional, organizational and geographic boundaries. Distributed requirements enhance the needs for achieving parallelism in the RE process by distributing requirements efforts among multiple independent organizations, supporting outsourcing of RE tasks, exploiting time zones differences to improve the process performance, managing/negotiating distributed teams of stakeholders and narrowing communication gaps among diversified stakeholders.

In this context, we can expect the development of more advanced collaborative tools to support the RE process for distributed requirements. Recent tool developments (e.g., [23]) indicate the importance for further R&D work in this area.

Additional focus on requirements reuse for software product lines

Software product lines/families have received a great adoption mainly in the embedded systems industry, as well as in other application domains. Due to their success, product families are characterized by a broadening of their scope and require different approach than conventional integration-oriented approaches.

Failing to identify the scope, commonalities/variabilities and reusable requirements artefacts of a product line, results in unresponsiveness of the platform, difficulties related to dealing with cross-cutting features and architectural challenges due to over-engineering.

In this context, we can expect advances in requirements reuse techniques for software product lines which will further improve/validate existing interesting solutions (such as multi-level feature trees [24] and hierarchical/composition-oriented product families [25]).

Research in more synergetic RE approaches

Field surveys regarding the current practice of RE reveal that a rather small subset of the available RE techniques is actually utilized by requirements engineers/designers. We believe that there possibilities for further research and empirical study towards more integrated methodologies which interrelate available methods/technologies into coherent, systematic and synergetic engineering processes.

Such a research orientation will result in well-defined approaches which will certainly improve the productivity, the quality and the adoption of RE in industry settings. There has been some promising work in this research “hotspot” (e.g., [26]) which shows that further research is expected on how to integrate various RE methods/technologies and propose effective methodologies.

More “sustainability” oriented RE

The process of eliciting and describing software requirements is, nowadays, more tightly related to the goal of attaining sustainability in the global environment.

According to the editorial note of the special issue on the IEEE Int. Req. Eng. Conf. RE’08 [27] there are two new challenges for RE:

“*First, when the target domain of software to be developed is directly/indirectly involved in the environmental issues, complex factors affecting the global environment should considered when analyzing and defining requirements. Second, only by fully understanding stakeholders’ needs, and documenting them in concise/unambiguous ways, we can consistently deliver quality products designed to meet the complexities of the advanced information society. Failing in engineering high quality requirements will surely bring in an unsustainable information society*”.

Last but not least, evaluation/empirical studies and benchmarking on how well research results cope with industrial problem domains are always beneficial for practitioners in order to analyze the advantages and disadvantages of each proposed approach and get help on selecting from among the various solutions. Therefore, empirical research studies are also expected to have a great impact on RE in the near future.

PART I: References

[1] Nuseibeh, B., & Easterbrook, S. (2000). Requirements engineering: a roadmap. In Proc. of the IEEE International Conference on Software Engineering (ICSE), pp. 35–46.

[2] Zave, P. (1997). Classification of research efforts in requirements engineering. ACM Computing Surveys, 29(4), pp. 315-321.

[3] Cheng, B.H.C., & Atlee, J. M. (2007). Research directions in requirements engineering. In Proc. of the IEEE Conference on Future of Software Engineering (FOSE'07), pp. 285–303.

[4] Lindquist, C. (2005). Fixing the requirements mess. CIO Magazine, pp. 52-60.

[5] Glass, R. L. (2005). IT failure rates – 70% or 10-15%?. IEEE Software, 22(3), pp. 110-112.

[6] Peterson, D. K., Kim, C., Kim, J. H., & Tamura, T. (2002). The perceptions of information systems designers from the United States, Japan, and Korea on success and failure factors. International Journal of Information Management, 22(6), pp. 421-439.

[7] Kappelman, L. A., McKeeman, R., & Zhang, L. X. (2006). Early warning signs of IT project failure: The dominant dozen. Information Systems Management, 23(4), pp. 31-36.

[8] McManus, J., & Wood-Harper, T. (2007). Understanding the sources of information systems project failure. Management Services, 51(3), pp. 38-43.

[9] Emam, K. E., & Koru, A. G., (2008). A replicated survey of IT software project failures. IEEE Software, 25(5), pp. 84-90.

[10] Chaos Report, Standish Group, 1994.

[11] Hansen, S., Berente, N., & Lyytinen, K., (2009). Requirements in the 21st century: current practice and emerging trends. In Lyytinen, Loucopoulos, Mylopoulos,

Robinson eds, Design Requirements Engineering: A Ten-Year Perspective. Lecture Notes in Business Information Processing, Vol. 14, Spinger-Verlag, pp. 44-86

[12] Capers, J., (2008). Applied software measurement. McGraw-Hill.

[13] Jarke, M., Loukopoulos, P., Lyytinen, K., Mylopoulos, J., & Robinson, W., (2009). High-impact requirements for software-intensive systems: a manifesto. Informatik-Spektrum, 32(4(2009), pp. 352-353 (full version in Jarke, M., Lyytinen, K., Mylopoulos, J. (eds.): Science of Design - High Impact Requirements Engineering for Software-Intensive Systems. Dagstuhl Seminar Proceedings, http://drops.dagstuhl.de/portals/index.php?semnr=08412)

[14] Jarke, M., Loukopoulos, P., Lyytinen, K., Mylopoulos, J., & Robinson, W., (2009). High-impact requirements for software-intensive systems: a manifesto. Informatik-Spektrum, 32(4(2009), pp. 352-353 (full version in Jarke, M., Lyytinen, K., Mylopoulos, J. (eds.): Science of Design - High Impact Requirements Engineering for Software-Intensive Systems. Dagstuhl Seminar Proceedings, http://drops.dagstuhl.de/portals/index.php?semnr=08412)

[15] Bergman, M., King, J.L., & Lyytinen, K., (2002). Large-Scale Requirements Analysis Revisited: The need for Understanding the Political Ecology of Requirements Engineering. Requirements Engineering, 7(3), pp. 152-171.

[16] Cockburn, A., (2006). Agile software development: the cooperative Game. 2nd Edition, Addison-Wesley.

[17] Pettersson, F., Ivarsson, M., Gorsheck, T., & Ohman, P., (2008). A practitioner's guide to lightweight software process assessment and improvement planning. Journal of Systems and Software, 21(6), pp. 972-995.

[18] CMMI Product Team, (2002). CMMI for systems engineering/software engineering / integrated product and process development/supplier sourcing. CMU/SEI-2002-TR-011.

[19] El Emam, K., Drouin, J. N., & Melo, W., (1998). SPICE: The theory and practice of software process improvement and capability determination. IEEE Press.

[20] Napier, N. P., & Mathiassen, L. (2009). Combining perceptions and prescriptions in requirements engineering process assessment: an industrial case study. IEEE Transactions on Software Engineering, 35(5), pp. 593-606.

[21] Zhou, Z., Zhang, J., McKinley, P. K., & Cheng, B. H. C., (2006). TA-LTL: Specifying adaptation timing properties in autonomic systems. In Proc. of the IEEE Workshop on Engineering of Autonomic and Autonomous Systems, (EASe 2006), pp. 109-118.

[22] Goldsby, H. J., Knoester, D.B., Cheng, B. H.C., McKinley, P. K., & Ofria, C. A. (2007). Digitally evolving models for dynamically adaptive systems. In Proc. of the 2007 International Workshop on Software Engineering for Adaptive and Self-Managing Systems, p.13.

[23] Damian, D., & Moitra, D., (eds.) (2006). Global software development. IEEE Software, Special issue, 23(5), 2006.

[24] Reiser, M. O., & Weber, M., (2006). Managing highly complex product families with multi-level feature trees. In Proc. of the 14th IEEE International Requirements Engineering Conference (RE'06), pp. 146-155.

[25] Bosch, J., (2006). The challenges of broadening the scope of software product families. Communications of the ACM, 49(12), pp. 41-44.

[26] Broy, M., (2006). The 'Grand Challenge' in informatics: engineering software-intensive systems. IEEE Computer, 39(10), pp. 72-80.

[27] Tamai, T. (2009). Introduction to the RE’08 special issue: requirements engineering for a sustainable world. Requirements Engineering, 14(4), pp. 229-287.

Part II: State of the Art in Software Component Adaptation and Reuse

1. Introduction to Component Adaptation & Relevant Benefits for SMEs

The first part of the deliverable has presented, in general, the state of the art in the Requirements Engineering processes, methodologies and methods/tools. In the second part of the deliverable we will concentrate on the availability of appropriate methods and tools that are suitable for Small and Medium Sized Enetrprises (SMEs) in order to reuse, after (possibly) adapting, software components in their product lifecycle, and, particularly, in their RE processes.

Reuse is essential for SMEs since it allows them to reduce the development expenses and achieve benefits that would be otherwise impossible to achieve. Especially the existence of freely usable and reusable open source software makes reuse a realistic approach for a RE process. Therefore, component selection, adaptation and reuse constitute areas of research interest in the context of the SPRINT SMEs project.

Every software product, however, has asscociated with certain requirements:

i) Functional requirements: these requirements specify what the product should provide. The services provided by the software product are implemented with software components. These components can be built from scratch or reused if suitable components exist. Usually though, although suitable components may exist that cover part of the functional requirements they require adaptation to fit exactly the functional requirements of a give software product.

ii) Quality (or non-functional requirements): Even in the rare case that a component fits exactly the functional requirements it still possible to be unusable in a given software product due to incompatibilities to quality requirements (e.g. the component may be too slow for a given application, or it may lack some security features that are required). In such cases the component will also need adaptations.

It seems therefore that component adaptation is a very important issue to achieve reuse of software components that is so important for the Greek software development SMEs.

2. Component Adaptation: Basic Concepts

Before giving a component adaptation definition, we start by defining what we mean by the term component, because the term may mean different things to different application domains (i.e. embedded systems, enterprise systems etc.).

Here we adopt Kruchten's definition of the term component which is given in [1]:

"*A component is a nontrivial, nearly independent, and replaceable part of a system that fulfils a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces*".

A component therefore is closely related to the system architecture and if it was not built specifically for a given architecture and its corresponding architecture requirements (functional and non-functional), then it must be adapted for reuse in a different architectural setting. To adapt a component it is often necessary to modify it and a categorization of components according to their modification potentiality becomes relevant.

In [2] the authors propose such a categorization which classifies components in two dimensions:

i) the source dimension (where the component came from), and

ii) the modification dimension ("*the degree to which a user (typically a system integrator) either can or must change the component's code*").

In the source axis we have the following categories:

S1. Independent commercial item

S2. Custom version of a commercial item

S3. Component produced to order under a specific contract

S4. Existing component obtained from external sources (for example, a reuse repository)

S5. Component produced in-house

In the modification axis we have the following categories:

M1. Very little or no modification

M2. Simple parameterization

M3. Necessary tailoring or customization

M4. Internal revision to accommodate special platform requirements

M5. Extensive functional recoding and reworking

One of the benefits of this categorization is that it allows us to consider the availability of the components' source code and its effect on the modifications that the reusers of the component can apply to it in order to reuse it. A software development SME's components will fall mainly to the S5 category (component produced in-house) as well as in the S4 category (existing components obtained from an external source).

We restrict our attention here to components of which the source code is available. This is the most interesting scenario for the Greek software development SMEs which will more often restrict themselves in reusing externally produced Free Open Source software components or in-house components due to the small cost of this approach. In both cases their source code will be available and therefore we can apply all the modification schemes of the modification axis (M1-M5). Open source software uniquely provides this flexibility which is very important.

Software adaptation is defined in [3] as "*the sequence of steps performed whenever a software entity is changed in order to comply with requirements emerging from the environment in which the entity is deployed*" and component adaptation in particular is described (again in [3]) as the "*the sequence of the steps required to bridge a component mismatch*".

Furthermore, the authors of [3] distinguish among three different types of adaptation:

i) Requirement adaptation which takes place during requirements engineering,

ii) Design-time adaptation which takes place during design, and

iii) Runtime adaptation which is specific to context-aware systems and takes place during system's runtime.

Here we will consider requirements and design-time adaptation to adapt components of which the source code is available (in-house or open source) for reuse from the SMEs according to their architectural requirements.

3. Component Adaptation Methods

The term “*Component Mismatch*” can refer to many different things and one should consider all these different types of mismatches for component adaptation. For example [4] is the first article that introduced the term *“Architectural Mismatch”* to describe the situation where components that seem ideal for certain purposes are used together in a software product, but their composition fails because these components don’t fit well together.

Architectural mismatches do not include low-level incompatibilities such as programming languages, operating systems and database schemas. It includes a more pervasive class of problems that “stem from mismatched assumptions a reusable part makes about the structure of the system it is to be part of”. They categorize the architectural mismatch problems in four categories:

i) The Nature of Components: assumptions

- On the infrastructure (both required and provided),
- On the control model (which component controls the sequencing of the computation) and
- On which component is responsible for managing data that is manipulated by the environment (data model),

ii) The Nature of Connectors: assumptions

- On the patterns of interactions (e.g. event broadcast, request/reply) and
- On the actual data exchanged between components through their connectors,

iii) The Global Architectural Structure: assumptions

- On the topology of system communications and
- On the presence or absence of certain components and connectors, and

iv) The Construction Process: assumptions

- On the order of pieces instantiations and
- On their combination in the system.

In order to handle these types of mismatches, something more is required than the usual information supplied by the syntactic interface of the component and the documentation that is usually available with a component. Especially in the case of the software development SMEs, which is the focus of the SPRINT-SMEs project, the lack of documentation is something that is observed very often both for in-house developed components as well as open source software components.

Component adaptation has been the subject of many research efforts in the field of Component-Based Software Engineering (CBSE). Many approaches to component adaptation are formal and consider the issue of interoperability problems with component interactions which they try to solve using a mathematical foundation [5, 6, 7].

These approaches have however many disadvantages, including:

i) They are perceived as complicated by the practitioners which are not experts in mathematics.

ii) They usually consider only a subset of interoperability problems (e.g. concurrency issues) and neglect the other problems that may exist.

iii) They assume the existence of appropriate specifications, which are not available most of the times and the effort to develop them for real-world software components very often outweighs the benefits of reuse.

iv) They are hard to scale to realistic software sizes.

Recently component adaptation has become one of the issues in the broader field of variability of Software Product Lines (SPLs) [8]. Software product lines however are not considering just component adaptation but adaptation of all artefacts of the software engineering process. Other component adaptation works include Composite

Components [9], superimposition [10], design patterns [3, 11], architectural tactics [12] and aspects [13].

Although these works are more applicable in practice and they try to address the issue of component adaptation from a technical perspective, they are not amenable to automated or generative transformations. Design patterns are an obvious solution to the adaptation of components, however, their application in an automated model-driven framework remains a research challenge although some works attempt to work toward this direction [14].

From the aforementioned approaches, the approach described in [3] is a very interesting approach which is also in line with SPRINT-SMEs researchers' own previous efforts to apply architectural tactics [12] and design patterns [11] for component adaptation. In this work the authors propose an engineering approach to component adaptation which distinguishes between the different mismatch types between components. These mismatch types include mismatches on signature (i.e. syntax), assertions, protocols, quality attributes and finally concepts. To adapt components with these types of mismatches the authors propose that the engineering approach should include several interface models. The engineering approach should have a syntactic interface, but in addition also include a behaviour interface (pre and post conditions), a synchronization interface, a quality of service interface and a concept interface.

Since (according to Meyer [15]) the assertion implication is difficult to decide and integrated with the other checks performed by the compiler, the authors decided that this type of checking would not be part of their engineering approach. The QoS interface is based on the ISO-9126 [16] standard for quality which includes a broad array of quality categories (although in their example they use QML [17] as a quality contract notation, since ISO-9126 does not have the notion of a contract). After detecting component mismatches using their interface models, the authors propose that design and architectural patterns can be applied to adapt them so that they can work together.

Their proposed approach comprises five steps:

i) Detection of mismatches: this directly depends on the availability of specifications,

ii) Selection of measures to overcome the mismatch: the authors comment that this also depends on the availability of specifications, however it is possible to filter out unsuitable patterns in advance,

iii) Configuring the measure: this means fine-tuning the application of the measure (i.e. pattern in the solution domain),

iv) Predicting the impact after applying the measure, and

v) Implementing and testing the solution.

It is worth noticing in this approach that the mismatch detection step assumes the existence of appropriate specifications, which is also partially required for the measure selection step. However, in general, software lacks such specifications, as we have already mentioned. This lack of specifications severely prohibits the reuse of open source software components and suggests that an engineering approach for SMEs should be relaxed and be less formal and more based on the practical availability of existing artefacts (e.g. test suites).

4. Component Adaptation Tools

Several tools have been constructed for component adaptation, using different underlying techniques such as Model-Driven Software Development, aspects, formal methods, design patterns etc.

Model-driven software development (MDSD) [18] is a new set of technologies that can be used for the generative approach to system development. It can be used however also as a method to adapt software components for reuse in different systems through the generation of wrappers, adapters or even component platforms.

For example, [19] presents a Model-Driven Software Development (MDSD) approach for component-based embedded systems. Instead of concentrating on code generation for components themselves this work takes a different approach in that it uses MDSD for generating component containers. A generative approach for container generation with the use of aspect is also discussed in [20]. The semi-automatic adaptation of components using design patterns to improve their quality is also an active research area with some initial promising results.

For example, [14] proposed triples to formally represent design patterns. The triples are of the form (MP, MS, T) in which MP (Problem Model) is a metamodel describing the problem, MS (Solution Model) is a metamodel describing the solution and T (Transformation rule) is a transformation that transforms instance models of MP to instance models of MS according to rules. The approach first works on marking input models with roles of the pattern metamodel and then applying transformation rules to the so-marked models to produce transformed models. The transformed models are instances of the solution metamodels (MS) corresponding to the problem metamodels (MP) of the design pattern applied. The authors provide an example of the Visitor Pattern (from the GoF book). MPs are metamodels in the sense that their instances are models of the problem domain.

Additionally to metaclasses, meta-associations and so on, MPs also include constructs such as evolution hotspots, which represent design problems that design patterns should solve. An example is a '++' symbol in an association's cardinality, which represents that additional objects could be added to this association in the future, thus signifying an evolution constraint. Similarly to MPs MSs are also represented as metamodels of the proposed solution. To enable the transformation of MPs to MSs in a given model, transformation rules are applied. A transformation can be considered as a rule in which the LHS is the MP and the RHS is the MS. However the application of the rule as a whole is to complex and not reusable and the authors propose two heuristics for breaking up rules to smaller more simple parts. The first heuristic handles the RHS and the second heuristic the LHS.

Formal approaches to component adaptation have also been supported by verification tools. In [21] a method is described for a hybrid approach to component composition which uses both architecture analysis and code synthesis. The approach is tool-based and automatic. First the architecture of the system is analyzed using the CHARMY tool and proven correct. Then the SYNTHESIS tool exploits the system architecture to perform local adaptations for each component so that reusable components (e.g. COTS, open source components etc.) can be used with confidence to this given software architecture. CHARMY generates a PROMELA model of the software architecture which is model checked by the SPIN model checker. Another formal approach to component adaptation which uses the theorem prover Atelier-B (http://www.atelierb.eu/en/) of the B formal method [22], is described in [7].

Integrated Development Environments also provide various levels of component adaptation. Almost all development environments support code refactoring [23], which is a quality improving technique that can be applied to source code. The Eiffel studio (http://www.eiffel.com) provides the developers with the capability to add Eiffel contracts to existing .NET assemblies and COM components and to migrate a C or C++ application from Unix to .NET.

5. The Significance of an Architectural Approach to Component Adaptation

Although several methods and tools for component adaptation already exist, as is evident from the discussion so far, most of them concentrate in syntactic adaptations, or successfully handle a subset of the possible component mismatches. Mismatches however can belong to several categories: syntax, behaviour, synchronization, quality of service and conceptual semantics [3, 4, 24]. A holistic approach to component adaptation that can handle all the above-mentioned types of mismatches, is therefore an open issue and we believe strongly that an approach like that should be based on architectural analysis.

But what exactly is architecture and related terms (e.g. architectural styles etc.)? The term "*architecture"* refers to the runtime decomposition of the system under study with the use of computational elements, the *components*, and their communication channels, the *connectors* [25].

The term "*architectural style*" refers to an abstract architecture where components, connectors, system properties etc. form the basis for the creation of concrete architectures that are compatible with the architectural style. In that respect, the relationship between the style and the architecture is somewhat similar to that of a class and its objects.

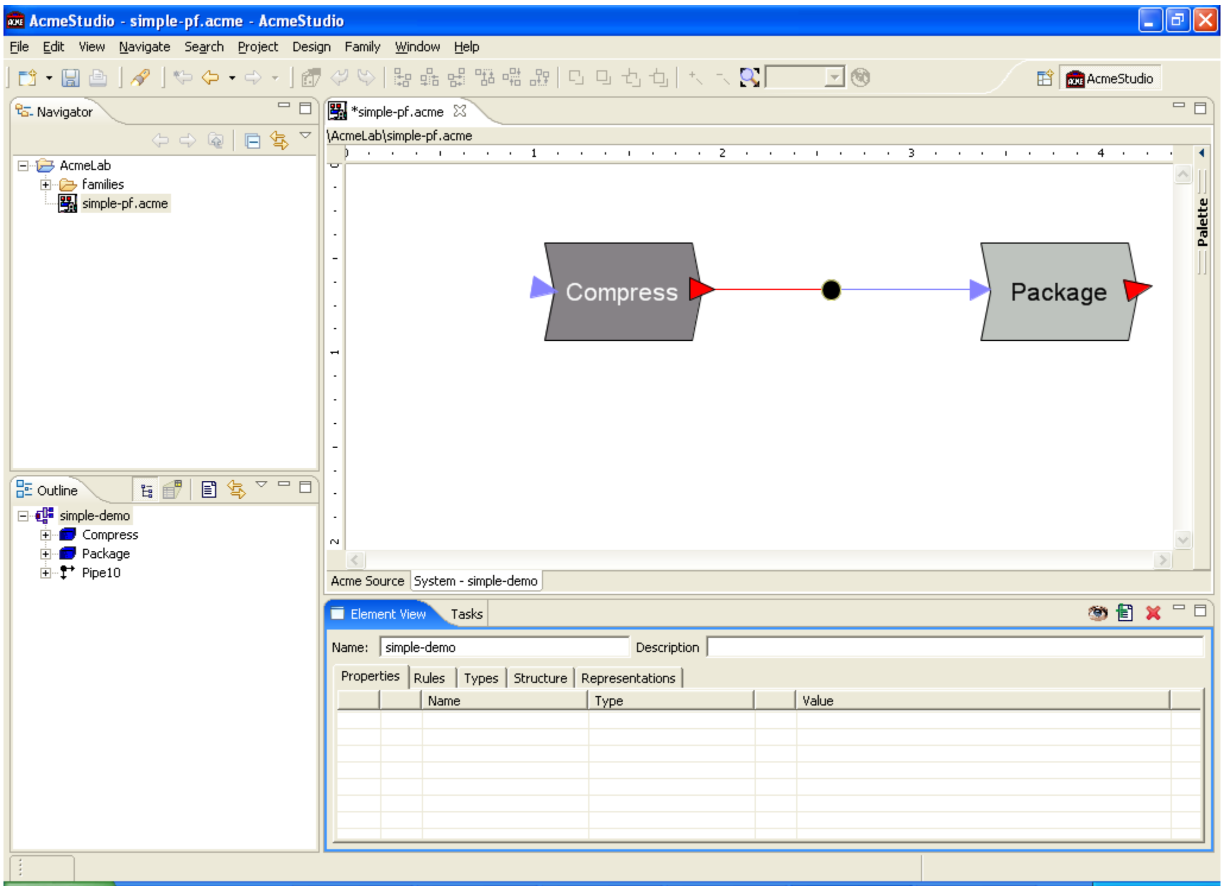
In the last decade several Architecture Description Languages (ADL) [26] have been proposed for the formal definition of architectures and architectural styles. The motivation for the use of formal ADLs is that the description of a system architecture with only boxes and lines drawings, does not provide a sound basis for analysis (e.g. whether an actual component implementation conforms to an architectural design and can be reused safely or not and required adaptation). It is difficult therefore, if not impossible, to foresee problems with components mismatches and the architectural requirements.

On the other hand, ADLs can be based on graphical notations which makes them more accessible to practitioners than mathematical notations mentioned earlier. Some ADLs are Aesop, Adage, C2, Darwin, Wright, Acme etc. All these ADLs share a common conceptual foundation with components, connectors, systems, properties, constraints and styles.

Next, we provide a description of the Acme ADL of the Carnegie Mellon University Software Engineering Institute as a typical example.

6. The Acme ADL

Acme is a tool-supported ADL by Acme Studio (http://www.cs.cmu.edu/~acme/AcmeStudio/), a plug-in of the popular Eclipse platform [27]. In Figure 1, a screenshot from Acme Studio is presented.



Name: simple-demo Description

Figure 1: Acme Studio: An environment for defining Acme architectures

With tools such as AcmeStudio an SME can adopt an architectural approach to component adaptation and reuse without resorting to precise mathematical approaches that are inaccessible to practitioners and have scalability issues to large real-world models.

With Acme the structure of the architecture is defined using seven types of entities: *components*, *connectors*, *ports*, *roles*, *representations* (more detailed low-level descriptions of components and connectors) and *representation maps* (rep-maps for short) that map the representations to their components or connectors). The static structure is augmented with properties.

Properties in Acme are auxiliary information such as *request-rate : float = 17.0*. Properties have no intrinsic semantics; they are uninterpreted values, useful for analysis, translation, display and manipulation with tools.

Design constraints are attached to model elements as *invariants* or *heuristics*. The difference between these two is that invariants are constraints that cannot be violated. Heuristics on the other hand are constraints that may be selectively violated but are observed. Invalidation of an invariant renders the whole architecture invalid, whereas invalidation of a heuristic is treated as a warning. Constraints are expressed as first order predicate logic expressions including also some specific functions from the architectural domain (such as the Boolean function connected (client, server) which is true if the client and server components are connected directly via a connector and false otherwise). Constraints determine the allowable ways of architecture evolution over time and they are scoped by the design element in which they are defined in an Acme system definition.

The formalization of *architectural styles* in Acme is achieved with the use of a type system for components, connectors, ports and roles which are used for the definition of recurring structures and relationships. An Acme style or family is defined by specifying a set of types and a set of constraints on these types that determine how the instances of the types can be used. In Acme types are interpreted as predicates and the system type as a conjunction of these predicates. Asserting the conformance of an instance to a style is then a process of asserting that it satisfies the predicate denoted by the type. This approach offers three advantages: it is possible for an architecture to conform to many styles, even to those that it was not declared to be an instance of. The use of invariants ties better with the type system because an invariant addition is done with the conjunction of its predicate to the other predicates of the type. Type checking is reduced to a process of checking the satisfaction of a set of predicates over a declared structure. The fact that, in general, checking for the satisfaction of predicates is not decidable and requires the use of a theorem prover, does not apply to Acme, because in Acme structures contain only a finite set of elements (components, connectors etc.).

The formalization of *architectural behavior* is achieved with the *Wright* architecture specification language. Formalization of architectural behavior allows us to specify and analyze architectural behavior. For example it allows us to say things such as that a pipe provides buffered and order-preserving data transmission and that service invocation in a component from a client component is done with a specific order. After specifying behavior we can analyze a system for deadlocks, race conditions, interface incompatibilities and other important properties. Wright language is a subset of Tony Hoare's *CSP* (Communicating Sequential Processes) [28, 29].

7. Conclusions of Part II: Overview in Component Adaptation/Reuse

In the second part of the deliverable we have first examined the need for component reuse for software development SMEs. Since reuse very often requires component adaptation for components that do not satisfy exactly the functional and non-functional (usually architectural) requirements, we examined the possible approaches and tools for component adaptation. We concluded that one of the most accessible approaches for software development SMEs is an architectural approach that uses ADL analysis tools to verify the existence of mismatches and regular tests, refactorings and design patterns at the source code level to adapt existing software components to new requirements in cases when the source code is available.

PART II: References

[1] P. Kruchten, The Rational Unified Process: An Introduction, Addison-Wesley Professional, 2000.

[2] D. Carney and F. Long, “What Do You Mean by COTS? Finally, a Useful Answer”, IEEE Software, vol. 17, Apr. 2000, pp. 83-86.

[3] S. Becker, A. Brogi, I. Gorton, S. Overhage, A. Romanovsky, and M. Tivoli, “Towards an Engineering Approach to Component Adaptation”, Architecting Systems with Trustworthy Components, Springer, 2006, pp. 193-215.

[4] D. Garlan, R. Allen, and J. Ockerbloom, “Architectural Mismatch: Why Reuse is So Hard”, IEEE Software, vol. 12, Nov. 1995, pp. 17-26.

[5] L. Alfaro and T. Henzinger, “Interface automata”, 9th ACM SIGSOFT international symposium on Foundations of Software Engineering (FSE'01), ACM Press, 2001, pp. 109-120 .

[6] Bracciali, A. Brogi, and C. Canal, “A formal approach to component adaptation”, Journal of Systems and Software, vol. 74, Jan. 2005, pp. 45-54.

[7] Mouakher, A. Lanoix, and J. Souquières, “Component Adaptation: Specification and Verification”, 11th International Workshop on Component Oriented Programming (WCOP 2006), ECOOP 2006, Nantes, France: 2006.

[8] F. Bachmann and P. Clements, Variability in Software Product Lines, 2005.

[9] S. Göbel, “Encapsulation of structural adaptation by composite components”, 1st ACM SIGSOFT workshop on Self-managed systems, Newport Beach, California : ACM Press, 2004, pp. 64 - 68.

[10] Bosch, “Superimposition: a component adaptation technique”, Information and Software Technology, vol. 41, Mar. 1999, pp. 257-273.

[11] G. Kakarontzas, P. Katsaros, and I. Stamelos, “Elastic Components: Addressing Variance of Quality Properties in Components”, Euromicro 2007 - CBSE Track, Lubeck, Germany: IEEE, 2007, pp. 31-38.

[12] G. Kakarontzas and I. Stamelos, “A Tactic-Driven Process for Developing Reusable Components”, 9th International Conference on Software Reuse (ICSR'9), Turin, Italy: Springer, 2006, pp. 273-286.

[13] Tešanovic, D. Nyström, J. Hansson, and C. Norström, “Aspects and Components in Real-Time System Development: Towards Reconfigurable and Reusable Software”, Journal pf Embedded Computing, vol. 1, Oct. 2004.

[14] G. Boussaidi and H. Mili, “A model-driven framework for representing and applying design patterns”, 31st Annual International Computer Software and Applications Conference (COMPSAC 2007), IEEE, 2007, pp. 97-100.

[15] B. Meyer, Object-Oriented Software Construction (2nd Edition), Prentice Hall PTR, 2000.

[16] ISO, “Software Engineering - Product Quality - Part 1: Quality Model”, 2001.

[17] S. Frolund and J. Koisten, “QML: A Language for Quality of Service Specification”m 1998.

[18] T. Stahl and M. Voelter, Model-Driven Software Development: Technology, Engineering, Management, Wiley, 2006.

[19] M. Voelter, C. Salzmann, and M. Kircher, “Model Driven Software Development in the Context of Embedded Component Infrastructures”, Component-Based Software Development for Embedded Systems, Springer, 2005, pp. 143-163.

[20] G. Moreno, “Creating custom containers with generative techniques”, 5th International Conference on Generative Programming and Component Engineering, Portland, Oregon, USA : ACM Press, 2006, pp. 29-38.

[21] Bucchiarone, A. Polini, P. Pelliccione, and M. Tivoli, “Towards an architectural approach for the dynamic and automatic composition of software components”, Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis, Portland, Maine: ACM, 2006, pp. 12-21.

[22] J-R. Abrial, The B-Book: Assigning Programs to Meanings, Cambridge University Press, 1996.

[23] T. Mens and T. Tourwe, “A survey of software refactoring”, IEEE Transactions on Software Engineering, vol. 30, Feb. 2004, pp. 126 - 139 .

[24] Beugnard, J. Jezequel, N. Plouzeau, and D. Watkins, “Making Components Contract Aware,” Computer, vol. 32, Jul. 1999, pp. 38-45.

[25] D. Garlan: “Formal Modeling and Analysis of Software Architectures: Components, Connectors and Events”, SFM 2003, LNCS 2804, pp. 1-24, 2003.

[26] N. Medvidovic and R. N. Taylor, “A Classification and Comparison Framework for Software Architecture Description Languages,” IEEE Transactions on Software Engineering, vol. 26, no. 1, pp. 70–93, Jan. 2000.

[27] D. Garlan, R. T. Monroe, and D. Wile: “Acme: Architectural Description of Component-Based Systems”, in Foundations of Component-Based Systems, Gary T. Leavens and Murali Sitaraman (editors), Cambridge University Press, 2000, pp. 47-68.

[28] C.A.R. Hoare: “Communicating Sequential Processes”, Prentice Hall Intl., 1985.

[29] A.W. Roscoe: “The Theory and Practice of Concurrency”, Prentice Hall, 1997.