2. Scientific part

2.1. Summary and key-words

All successful software systems evolve to meet changing requirements. Without continual reengineering, however, such systems necessarily suffer from *architectural drift*, as their original design no longer matches new business goals and requirements. As a consequence, they become increasingly complex and fragile, leading to ever higher maintenance costs.

We propose a *component-based approach to software evolution*, in which stable software artifacts are identified over time, and are migrated towards components and component architectures. The key to the approach is a *component meta model* for modelling and analysing evolving software systems. This meta model facilitates *component migration* tools and techniques. As a successful software system matures, instead of becoming more complex and fragile, its architecture gradually migrates towards a configuration of software components, which can be more easily reconfigured and adapted than a typical legacy system. To achieve this flexibility, however, a component-based system requires a suitable *compositional infrastructure* for specifying component configurations and the compositional abstractions that hold the components together.

We propose to develop (i) a component meta model for modelling software systems that extends existing standards (such as UML) with concepts required to support evolution, focusing on such issues as non-functional requirements and software dependencies. Based on this meta model, we will develop (ii) component migration tools and methods that will help to identify candidate components, identify and resolve architectural and design drift, and support transformation to component-based software structures. We will focus on software metrics and visualization to support analysis, and language-independent refactorings to support transformation. Component migration methods will be documented as reverse and reengineering patterns. Finally, we propose to develop (iii) a compositional infrastructure to support architectural specification, and run-time configuration and evolution, using the agent-based framework of the Piccola composition language.

Keywords

Software components, software evolution, software reengineering, meta-modelling, software metrics, visualization, software architecture, software composition.

2.2. Research plan

2.2.1. State of the art and related work

Component Meta Model

Industry has converged in the last few years on the Unified Modelling Language (UML) as a standard notation for expressing object-oriented models [3]. UML defines its own meta model, which can be extended to and adapted to various needs. Any serious meta modelling research effort must therefore position itself with respect to UML.

UML suffers from two serious drawbacks for component migration. First, UML is conceived as language for specifying analysis and design models, and is missing concepts needed for describing and working with implementation models. Fine-grained relationships between individual code elements, for example, can only be modelled by extending UML in non-standard ways. Second, UML is very weak when it comes to expressing non-functional properties (such as quality-of-service, resource consumption or real-time constraints) or expressing compositionality dependencies and constraints (such as protocols that must be respected when using an interface, or assumptions that must be respected when extending a component or overriding behaviour inherited by a subclass).

Considerable work has been done on real-time extensions to UML [10], and standardization efforts are under way, but little has been done so far on other kinds of non-functional requirements. The notion of *contracts* as explicit representations of contractual obligations between objects or components and their clients is well-established [15][24] but UML provides only an informal mechanism to express constraints (usually restricted to data modelling constraints). Various researchers have proposed enriching interfaces to express reusability constraints [20][34], but this is still an open research issue, and UML does not address it.

XML is clearly emerging as a standard for interchange of models [4][5]. XML is a successor of SGML, intended to express arbitrary models, not just document mark-up.

Until now there is no commonly agreed definition of what a component is [35][65]. Most researchers and authors agree that components should be “black box” entities, and preferably binary entities [35], but there is no agreement what constitutes a “black box” since even binary components may exhibit hidden platform dependencies. There is also considerable interest in source code components in the form of C++ templates. Moreover, multiple component models exist, such as EJB, COM+ and CORBA. Component repositories exist [9][16][25][37] but there is no uniform way to represent their basic structure and properties, and there is no agreement what properties need to be captured and represented.

In the past few years there has been increasing interest in abstracting not just components but common *architectures* from a set of related applications (i.e., a “product line”). Such “architectural styles” have been documented and specified using a range of experimental *architectural description languages*, with the goal of formalizing and reasoning about applications at an architectural level [32].

Component Migration

Traditional reengineering of legacy system promotes *wrapping* technologies where the legacy code is encapsulated within object-oriented languages like Java or Smalltalk [6]. Various techniques, like data flow analysis [7] may be used to decompose the legacy code into subsystems that can then be wrapped as components. *Component mining*, on the other hand, tries to identify generic software entities in legacy code and extract them as components. Several techniques are applied to identify candidate components [19] [14]: metrics [13], clustering algorithms [1], and concept analysis [21][33].

Once the analysis is done, code transformations are necessary. In the context of object-oriented programming, behaviour preserving code transformations are known as *refactorings* [28][30]. The Refactoring Browser is the best known tool currently available [30], but can only be applied to Smalltalk.

Compositional Infrastructure

Composition infrastructure is concerned first of all with how to *specify* compositions of components, and second with how to *effect* component interconnections at run-time. The first subject is current addressed by (i) advances in programming languages, (ii) scripting languages, (iii) graphical "builders" and 4GL environments. The second issue is largely addressed by so-called middleware platforms.

Purely functional programming languages have always been good at expressing (functional) composition, but do not explicitly address issues of concurrency and distribution which are critical for real component-based systems. Recently there has been some investigation into using functional languages to compose components [18].

C++ template meta-programming has become extremely popular in specialized domains where compile-time composition of software components is required to achieve adequate performance. In this approach, compile-time polymorphism is favoured over run-time polymorphism, but all polymorphism is statically resolved by means of compile-time reflection [8]. Since C++ templates are untyped, all type-checking is performed after composition is completed.

One of the key difficulties in run-time component composition is that not all interfaces can be known statically. Java solved this problem by providing a so-called "reflection" package (technically this is introspection, since Java can only inspect the class of an object, not change it at run-time).

Conventional programming languages typically provide no special mechanisms to support component composition. Instead, so-called *scripting languages* have come in vogue, which allow programmers to compose applications with high-level "scripts" which direct and coordinate components typically written in a conventional programming language. Languages like Visual Basic [11][26], Perl [36], TCL [29] and Python [23] have become very popular in recent years, especially in domains like internet programming, where applications are rapidly developed and evolved. Each of these languages is typically well suited for scripting some particular domain (Perl is good at text manipulation, TCL is good at GUI building), and not so good at others, thus leading to a proliferation of specialized scripting languages.

Similarly, graphical application builders and 4GL environments [22] are typically highly optimized for certain tasks, such as GUI construction or development of form-based database applications, but cannot be adapted to support graphical composition in other domains.

Run-time support for software composition is generally limited to middleware systems like CORBA [2][27] and COM [17][31], which provide mechanisms to lookup components and services, some degree of service negotiation, communication between distributed components and their clients, and marshalling of communicated values (if necessary). Components that are to be made available through such middleware systems must publish their interfaces with a so-called Interface Definition Language, or IDL. Such interfaces are limited to expresses service signatures — non-functional properties, dependencies, contracts, constraints and protocols cannot be expressed [12], but they are critical in order to avoid compositional mismatch [35]. In COM, for example, non-functional properties are specified at the object level, but not in the interfaces. Similarly, MTS (the Microsoft Transaction Server, now part of COM+) [11], provides transaction and security services for run-time components, but the service guarantees cannot be specified at the interface level.

[1] Nicolas Anquetil and Timothy C. Lethbridge, “Experiments with Clustering as a Software Remodularization Method,” *WCRE’99 (Sixth Working Conference on Reverse Engineering)*, 1999, pp. 235-256.

[2] Ron Ben-Natan, *Corba*, McGraw-Hill, 1995.

[3] Grady Booch , James Rumbaugh and Ivar Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 1998, ISBN: 0-210-57168-4.

[4] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen (eds), Extensible Markup Language (XML) 1.0, W3C Recommendation, http://www.w3.org/TR/1998/REC-xml-19980210, Feb 10, 1998.

[5] Dan Connolly, Extensible Markup Language (XML), http://www.w3.org/XML/, website, 1997-2000.

[6] Michael Brodie and Michael Stonebraker, *Migrating Legacy Systems: Gateways, Interfaces and the Incremental Approach*, Morgan Kaufman, 1995.

[7] R. Clayton , S. Rugaber and L. Wills, “Incremental Migration Strategies: Data Flow Analysis for Wrapping,” *Proceedings of WCRE’98*, IEEE Computer Society, 1998, pp. 69-79, ISBN: 0-8186-89-67-6.

[8] Krzysztof Czarnecki, Ulrich W. Eisenecker, “Components and Generative Programming”, ESEC / SIGSOFT FSE, 1999, pp 2-19.

[9] P. Devanbu, R.J. Brachman, P.G. Selfridge and B.W. Ballard, "LaSSIE: A Knowledge-Based Software Information System", Communications of the ACM 34, 5, pp. 34-49, 1991

[10] Bruce Powel Douglass, *Real-Time UML Second Edition*, Addison-Wesley, 1999.

[11] Guy Eddon and Henry Eddon, *Inside COM+*, Base Services, Microsoft Press, 1999.

[12] David Garlan, Robert Allen and John Ockerbloom, “Architectural Mismatch: Why Reuse Is So Hard,” *IEEE Software*, vol. 12, no. 6, Nov 1995, pp. 17-26.

[13] Jean-Francois Girard and Rainer Koschke, “A Metric-based Approach to Detect Abstract Data Types and Abstract State Encapsulation,” *Conference on Automated Sotfware Engineering*, 1997.

[14] Jean-Francois Girard and Rainer Koschke, “A Comparison of Abstract Data Type and Object Detection Techniques,” *Science of Computer Programming Journal*, Elsevier Science Publisher, 1999.

[15] Richard Helm, Ian M. Holland and Dipayan Gangopadhyay, “Contracts: Specifying Behavioural Compositions in Object-Oriented Systems,” *Proceedings OOPSLA/ECOOP’90, ACM SIGPLAN Notices*, vol. 25, no. 10, Oct. 1990, pp. 169-180.

[16] S. Henninger, "An Evolutionary Approach to Constructing Effective Software Reuse Repositories", ACM Transactions on Software Engineering and Methodology 6, 2, pp. 111-140, 1997

[17] Inprise Corporation, *Inside COM+*, C++ Builder 4 Developer's Guide, 1999

[18] Simon Peyton Jones, Erik Meijer and Daan Leijen, “Scripting COM components in Haskell,” *Fifth International Conference on Software Reuse*, Victoria, British Columbia, June 1998.

[19] Rainer Koschke, “An Incremental Semi-Automatic Method for Component Recovery,” *WCRE’99 (Sixth Working Conference on Reverse Engineering)*, 1999, pp. 256-267.

[20] John Lamping, “Typing the Specialization Interface,” *Proceedings OOPSLA ’93, ACM SIGPLAN Notices*, vol. 28, no. 10, Oct. 1993, pp. 201-214.

[21] Christian Lindig and Gregor Snelting, “Assessing Modular Structure of Legacy Code based on Mathematical Concept Analysis,” *Proceegins of ICSE’97*, 1997.

[22] Ray Lischner, *Secrets of Delphi 2*, Waite Group Press, 1996.

[23] Mark Lutz, *Programming Python*, O’Reilly, 1996.

[24] Bertrand Meyer, “Applying Design by Contract,” *IEEE Computer (Special Issue on Inheritance & Classification)*, vol. 25, no. 10, Oct. 1992, pp. 40-52.

[25] A. Michail and D. Notkin, "Assessing Software Libraries by Browsing Similar Classes, Functions and Relationships", In Proceedings of the 21st ICSE, (Los Angeles, CA), pp. 463-472, 1999

[26] Microsoft Corporation, *Visual Basic Programmierhandbuch*, 1997

[27] Object Management Group, The Common Object Request Broker: Architecture and Specification, July 1996

[28] William F. Opdyke, *Refactoring Object-Oriented Frameworks*, University of Illinois, 1992, Ph.D. Thesis.

[29] John K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.

[30] Don Roberts, John Brant and Ralph E. Johnson, “A Refactoring Tool for Smalltalk,” *Theory and Practice of Object Systems (TAPOS)*, vol. 3, no. 4, John Wiley & Sons, 1997, pp. 253-263.

[31] Dale Rogerson, *Inside COM: Microsoft’s Component Object Model*, Microsoft Press, 1997.

[32] Mary Shaw and David Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.

[33] Michael Siff and Thomas Reps, “Identifying Modules Via Concept Analysis,” *Proceedings of International Conference on Software Maintenance*, 1997.

[34] Patrick Steyaert, Carine Lucas, Kim Mens and Theo D'Hondt, "Reuse Contracts: Managing the Evolution of Reusable Assets," *Proceedings of OOPSLA '96 Conference*, ACM Press, 1996, pp. 268-285.
[35] Clemens A. Szyperski, *Component Software*, Addison-Wesley, 1998.
[36] Larry Wall and Randal L. Schwartz, *Programming Perl*, O'Reilly & Associates, Inc., 1990.
[37] A.M. Zaremski and J.M. Wing, "Specification Matching of Software Components", ACM Transaction on Software Engineering and Methodology 6, 4, pp. 333-369, 1997

2.2.2. Contributions to the field by the applicants

We have carried out related work both within the ongoing NFS project, "A framework approach to composing heterogeneous applications" (NFS 20-53711.98), and within FAMOOS, "A Framework-based Approach for Mastering Object-Oriented Software Evolution" (ESPRIT Project 21975, BBW Nr. 96.0015).

Component Meta Model

Within the NFS project we have carried out extensive modelling experiments to develop a formal semantics of software components and composition mechanisms. This work has led to (i) the πL Calculus, a process calculus foundation for software components in which concurrent agents communicate *forms*, or extensible records instead of tuples [61][62][74], (ii) an approach to modelling objects and components based on explicit meta-objects encoded in πL [60][76][72][73].

Within FAMOOS we have carried out a detailed analysis of UML and determined that it was not adequate for supporting reengineering operations [45]. As a consequence we defined a language independent meta model named Famix [47] and a corresponding interchange format [66]. Famix is currently the foundation of the Moose reengineering environment [53], which in turn is the basis for the Moose Refactoring Engine, CodeCrawler [44][59][49], and Moose/Metrics [43] and [48].

Using Moose, we have evaluated how size metrics can support the understanding of application evolution by means of recovering refactorings [48] and how size metrics can assess the quality of software entities [43]. With CodeCrawler, we have shown how the combination of simple metrics and simple graphs support the assessment of object-oriented applications [44][49][52][59]. We have also used the Famix model to evaluate the impact of changes between software versions on the technical documentation [55].

Component Migration

Within FAMOOS we have worked on the reengineering of object-oriented legacy applications. We have particularly focused on the first necessary steps of component migration: identification of duplicated code [50], language independent code refactorings, abstraction identification, supporting iterative model extraction, and architectural extraction [67][68][69][71][79]. We have also developed the following tools: DupLoc, a tool for detecting code duplication in a language independent manner [50]; Gaudi, an environment for supporting iterative extraction of architectural view based on logic programming [70]; and the Moose/Refactoring Engine, a language-independent refactoring engine.

At the methodological level, we have identified a series of guidelines for developing component frameworks [41][77], and we are also continuing to analyse the process of reengineering by recording best practice in reverse engineering and reengineering in terms of patterns [42][46][51]. We have also started to investigate how to support the refactoring process by means of special "assistants" that suggest transformations to be performed [54].

Compositional Infrastructure

Piccola is an experimental composition language [64] whose semantics is defined by a mapping to πL [38][39][63][75]. The current version of Piccola is implemented in Java, and translates Piccola scripts to πL, which is then executed by a πL interpreter.

Piccola can be used to define (i) architectural styles, such as push-flow or pull-flow pipes and filters, GUI composition, publisher-subscriber composition, and so on, (ii) coordination abstractions to mediate between concurrent activities, (iii) glue abstractions to bridge compositional styles, and (iv) scripts, which specify how external components are plugged together using composition, coordination and glue abstractions. The current implementation is performant enough to carry out non-trivial experiments, and we are now exploring some larger case studies. We expect that the formal foundation of Piccola will allow us to reason about compositions and their properties (for example, performance costs, or real-time guarantees) in a way that is not possible with scripting languages.

Piccola can presently be used only to compose components written in Java or Piccola, and there is no support yet for distribution using either middleware or coordination media. We have, however, previously explored the use of scripting languages, particularly Python, for scripting CORBA components, and for wrapping existing software to export services to CORBA [57]. We have also experimented with the development of reusable coordination abstractions for Java [78], form-based coordination media for distributed components [58], and mobile software components [56]. We expect these earlier experiments to lead to a more systematic approach to composition of distributed components based on Piccola.

We have also carried out some initial experiments in visualizing and controlling agent communication within a running Piccola system [40], and we believe these initial ideas can be elaborated to support run-time monitoring and reconfiguration of component-based software systems.

References in bold are provided in the annex to this proposal.

[38] Franz Achermann and Oscar Nierstrasz, "Applications = Components + Scripts — A tour of Piccola," Software Architectures and Component Technology, Mehmet Aksit (Ed.), Kluwer, 2000, to appear.

[39] Franz Achermann, Markus Lumpe, Jean-Guy Schneider and Oscar Nierstrasz, "Piccola - a Small Composition Language," *Formal Methods for Distributed Processing, an Object Oriented Approach,* Howard Bowman and John Derrick. (Ed.), Cambridge University Press., 2000, to appear.

[40] Cristina Gheorghiu Cris, "Visualisierung von pi-Programmen," Informatikprojekt, University of Bern, Jan. 99.

[41] Serge Demeyer, Theo Dirk Meijler, Oscar Nierstrasz and Patrick Steyaert, "Design Guidelines for Tailorable Frameworks," *Communications of the ACM*, vol. 40, no. 10, ACM Press, October 1997, pp. 60-64.

[42] Serge Demeyer, Matthias Rieger and Sander Tichelaar, *Three Reverse Engineering Patterns*, April, 1998, Writing Workshop at EuroPLOP'98.

[43] Serge Demeyer and Stéphane Ducasse, "Metrics, Do They Really Help?," *Proceedings LMO'99 (Languages et Modèles à Objets)*, Jacques Malenfant (Ed.), HERMES Science Publications, Paris, 1999, pp. 69-82.

[44] Serge Demeyer, Stéphane Ducasse and Michele Lanza, "A Hybrid Reverse Engineering Platform Combining Metrics and Program Visualization," *WCRE'99 Proceedings (6th Working Conference on Reverse Engineering),* Francoise Balmas, Mike Blaha and Spencer Rugaber (Ed.), IEEE, October, 1999.

[45] Serge Demeyer, Stéphane Ducasse and Sander Tichelaar, "Why Unified is not Universal. UML Shortcomings for Coping with Round-trip Engineering," *Proceedings UML'99 (The Second International Conference on The Unified Modeling Language),* Bernhard Rumpe (Ed.), LNCS 1723, Springer-Verlag, Kaiserslautern, Germany, October, 1999.

[46] Serge Demeyer, Stéphane Ducasse and Sander Tichelaar, "A Pattern Language for Reverse Engineering," *Proceedings of the 4th European Conference on Pattern Languages of Programming and Computing, 1999,* Paul Dyson (Ed.), UVK Universitätsverlag Konstanz GmbH, Konstanz, Germany, July, 1999.

[47] Serge Demeyer, Sander Tichelaar and Patrick Steyaert, "FAMIX 2.0 - The FAMOOS Information Exchange Model," technical report, University of Berne, August, 1999.

[48] Serge Demeyer, Stéphane Ducasse and Oscar Nierstrasz, "Finding Refactorings via Change Metrics," working paper, April, 1999.

[49] Stéphane Ducasse and Michele Lanza, “Towards a Reverse Engineering Methodology for Object-Oriented Systems,” *Techniques et Sciences Informatiques*, 2000, Submitted to Techniques et Sciences Informatiques, Edition Speciale Reutiliusation.

[50] Stéphane Ducasse, Matthias Rieger and Serge Demeyer, “A Language Independent Approach for Detecting Duplicated Code,” *Proceedings ICSM’99 (International Conference on Software Maintenance)*, Hongji Yang and Lee White (Ed.), IEEE, September, 1999, pp. 109-118.

[51] Stéphane Ducasse, Tamar Richner and Robb Nebbe, “Type-Check Elimination: Two Object-Oriented Reengineering Patterns,” *WCRE’99 Proceedings (6th Working Conference on Reverse Engineering)*, Francoise Balmas, Mike Blaha and Spencer Rugaber (Ed.), IEEE, October, 1999.

[52] Stéphane Ducasse, Michele Lanza and Serge Demeyer, “A Hybrid Reverse Engineering Approach Combining Metrics and Program Visualization,” In Object-Oriented Technology (ECOOP'99 Workshop Reader), LNCS (Lecture Notes in Computer Science), N 1800, Springer - Verlag, 1999.

[53] Stéphane Ducasse, “The Moose Environment: a First Documentation,” Technical Report, University of Berne, 1999.

[54] Stéphane Ducasse, Matthias Rieger and Georges Golomingi, “Tool Support for Refactoring Duplicated OO Code,” *Proceedings of the ECOOP’99 Workshop on Experiences in Object-Oriented Re-Engineering*, Stéphane Ducasse and Oliver Ciupke (Ed.), Forschungszentrum Informatik, Karlsruhe, June 1999, FZI-Report 2-6-6/99.

[55] Fredi Frank, “An Associative Documentation Model,” Diploma thesis, University of Bern, October 1999.

[56] Jürg Gertsch, “Fruitlets - a Kind of Mobile Component,” Diploma thesis, University of Bern, June 1997.

[57] Michael Held, “Scripting für CORBA,” Diploma thesis, University of Berne, March 1999.

[58] Daniel Kühni, “APROCO: A Programmable Coordination Medium,” Diploma thesis, University of Bern, October 1998.

[59] Michele Lanza, “Combining Metrics and Graphs for Object Oriented Reverse Engineering,” Diploma thesis, University of Bern, October 1999.

[60] Markus Lumpe, Jean-Guy Schneider and Oscar Nierstrasz, “Using Metaobjects to Model Concurrent Objects with PICT,” *Proceedings of Languages et Modèles à Objects*, Leysin, October 1996, pp. 1-12.

[61] Markus Lumpe, Jean-Guy Schneider, Oscar Nierstrasz and Franz Achermann, “Towards a formal composition language,” *Proceedings of ESEC ’97 Workshop on Foundations of Component-Based Systems*, Gary T. Leavens and Murali Sitaraman (Ed.), Zurich, September 1997, pp. 178-187.

[62] Markus Lumpe, “A Pi-Calculus Based Approach to Software Composition,” Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics, January 1999.

[63] Markus Lumpe, Franz Achermann and Oscar Nierstrasz, “A Formal Language for Composition,” *Foundations of Component Based Systems*, Gary Leavens and Murali Sitaraman (Ed.), Cambridge University Press, 2000, to appear.

[64] Oscar Nierstrasz and Theo Dirk Meijler, “Requirements for a Composition Language,” *Object-Based Models and Langages for Concurrent Systems*, P. Ciancarini, O. Nierstrasz and A. Yonezawa (Ed.), LNCS 924, Springer-Verlag, 1995, pp. 147-161.

[65] Oscar Nierstrasz and Laurent Dami, “Component-Oriented Software Technology,” *Object-Oriented Software Composition*, O. Nierstrasz and D. Tsichritzis (Ed.), Prentice Hall, 1995, pp. 3-28.

[66] Oscar Nierstrasz, Sander Tichelaar and Serge Demeyer, “CDIF as the Interchange Format between Reengineering Tools,” *OOPSLA’98 Workshop on Model Engineering, Methods and Tools Integration with CDIF*, October, 1998.

[67] Tamar Richner and Robb Nebbe, “Analyzing Dependencies to Solve Low-Level Problems,” *Object-Oriented Technology (ECOOP’97 Workshop Reader)*, Jan Bosch and Stuart Mitchell (Ed.), LNCS 1357, Springer-Verlag, June, 1997, pp. 266-267.

[68] Tamar Richner, “Describing Framework Architectures: more than Design Patterns,” *Proceedings of the ECOOP ’98 Workshop on Object-Oriented Software Architectures*, Jan Bosch, Helene Bachatene, Görel Hedin and Kai Koskimies (Ed.), Research Report 13/98, University of Karlskrona, July, 1998.

[69] Tamar Richner, Stéphane Ducasse and Roel Wuyts, “Understanding Object-Oriented Programs with Declarative Event Analysis,” *Object-Oriented Technology (ECOOP’98 Workshop Reader)*, Serge Demeyer and Jan Bosch (Ed.), LNCS 1543, Springer-Verlag, July, 1998.

[70] Tamar Richner and Stéphane Ducasse, “Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information,” *Proceedings ICSM’99 (International Conference on Software Maintenance)*, Hongji Yang and Lee White (Ed.), IEEE, September, 1999, pp. 13-22.

[71] Tamar Richner, “Using Recovered Views to Track Architectural Evolution,” *to appear in ECOOP’99 Workshop Reader*, Springer-Verlag, June, 1999.

[72] Jean-Guy Schneider and Markus Lumpe, “Modelling Objects in PICT,” Technical Report, no. IAM-96-004, University of Bern, Institute of Computer Science and Applied Mathematics, January 1996.

[73] Jean-Guy Schneider and Markus Lumpe, “Synchronizing Concurrent Objects in the Pi-Calculus,” *Proceedings of Langages et Modèles à Objets ’97*, Roland Ducournau and Serge Garlatti (Ed.), Hermes, Roscoff, October 1997, pp. 61-76.

[74] Jean-Guy Schneider, “Components, Scripts, and Glue: A conceptual framework for software composition,” Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics, October 1999.

[75] Jean-Guy Schneider and Oscar Nierstrasz, “Components, Scripts and Glue,” *Software Architectures — Advances and Applications*, Leonor Barroca, Jon Hall and Patrick Hall (Ed.), Springer, 1999, pp. 13-25.

[76] Jean-Guy Schneider and Markus Lumpe, “A Metamodel for Concurrent, Object-based Programming,” *Proceedings of Langages et Modèles à Objets 2000*, Christophe Dony (Ed.), Hermes, Montreal, January 2000, to appear.

[77] Sander Tichelaar, Juan Carlos Cruz and Serge Demeyer, “Design Guidelines for Coordination Components,” *Proceedings ACM SAC 2000 - Track on Coordination*, ACM Press, March, 2000, to appear.

[78] Sander Tichelaar, “A Coordination Component Framework for Open Distributed Systems,” Master’s Thesis - Software Composition Group, University of Groningen, NL - University of Berne, CH, May 1997.

[79] Sander Tichelaar, Stéphane Ducasse and Theo-Dirk Meijler, “Architectural Extraction In Reverse Engineering by Prototyping: An experiment,” *Proceedings of the ESEC/FSE Workshop on Object-Oriented Re-engineering*, Serge Demeyer and Harald Gall (Ed.), Technical University of Vienna, Information Systems Institute, Distributed Systems Group, September, 1997, Technical Report TUV-1841-97-10.

2.2.3. Detailed research plan

Component Meta Model

We propose to develop a component meta model that provides concepts required to support evolution, focusing on such issues as non-functional requirements and software dependencies. The meta model will allow models of software systems to be specified, analysed and manipulated.

The meta model will extend FAMIX to address architectural constraints pertinent to component systems, and, like FAMIX, will strive for maximal compatibility with UML. We plan to evaluate how the UML Meta-Object Facility (MOF) can be instantiated to support the description of components (i.e., from CORBA, EJB, COM+). With the resulting meta model, we expect to be able to express component requirements, constraints, contracts, and protocols.

Various tools will also be developed to support querying and analysis.

- The FAMIX *meta mode*l will be extended to address architectural constraints, such as non-functional requirements, service obligations and protocols, architectural style, and reusability and extensibility contracts.
- A *component repository* will be developed to store software models and act as a server for various tools.
- A *metrics rendering tool* will be developed to analyse and evaluate software systems. The tool will address issues currently not covered by Codecrawler, such as coupling and cohesion metrics, and comparisons between successive versions of software systems.
- An *interactive query system* will be developed to monitor and evaluate versions of evolving software systems. Both pre-packaged and ad hoc queries will be supported to help component developers identify which parts of a system are stabilizing, and which parts require additional flexibility.

Component Migration

Based on the component meta model, we will develop component migration tools to identify candidate components, identify and resolve architectural and design drift, and support transformation to component-based software structures. We will focus on software metrics and visualization to support analysis, and language-independent refactorings to support transformation.

Migration of applications towards component architectures requires both an iterative development process and suitable tools. We believe such a process can be expressed as a *reengineering pattern language*. We have already identified a large number of reengineering patterns that express "best practice" in iterative development of object-oriented systems, and hope to expand this set to a relatively complete system of patterns (known in the literature as a "pattern language") to define a process for migration towards components.

- *Software analysis tools* will be developed to detect architectural and design drift by exploiting static and dynamic information of the software models in the repository. Duplicated code is already detected by Duploc. Duplicated functionality may be detected by similar means, with the help of filtering techniques. Coupling and cohesion metrics can help to detect architectural drift. Various other metrics can be used to detect violation of other object-oriented and software engineering principles (such as tight data coupling, navigational code, or missing polymorphism). We especially plan to address:
  - — software evolution visualization (multiple versions of the same entity)
  - — very large scale extension (applications, packages, components)
  - — very small scale extension (internal structure of classes)
- *Migration assistants* will be developed that suggest software entities as candidates for abstraction or "componentization" based on the results of the analysis. These "assistants" will apply simple heuristics documented in the pattern language to suggest plausible refactorings to reduce architectural and design drift. For example, duplicated code can typically be factored out as hook methods, and poor coupling and cohesion can often be resolved by similarly simple refactorings. Strongly collaborating entities identified in dynamic analysis may be candidates for new software components.
- *Language independent refactoring tools* would automate the tedious task of abstracting, renaming and moving software entities, based on the component meta model, rather than on the concrete meta model of a specific programming language. Many of the transformations performed by the Refactoring Browser on Smalltalk code, for example, can be generalized to work with other object-oriented languages, such as C++ and Java.
- A *reengineering pattern language* will be defined that guides component developers in (i) reverse engineering, or recovering and understanding software models, (ii) detecting problems of flexibility and architectural drift, (iii) identifying new target software structures that solve these problems, and (iv) transforming the software to the new structures. Point (i) is supported by the component meta model and tools. Points (ii), (iii) and (iv) are further supported by the component migration tools.

Compositional Infrastructure

A component-based software system consists not only of generic software components, but also some specialized ones, tailored to the task, a set of *composition abstractions* that mediate the interconnections between components, a *configuration* which specifies those interconnections (and which may evolve at run-time), *middleware* which mediates between components on different platforms, and *glueware* which adapts components and bridges in-

terfaces and protocols of different component architectures. The composition language, Piccola, will be extended in various ways to address these concerns.

- A *middleware bridge* will be developed from Piccola to CORBA/COM+.
- A *distributed form space* will be developed to allow Piccola agents to coordinate components on networked systems.
- *Composition abstractions* for a variety of component architectures will be developed. Special attention will be given to compositional reasoning: architectural constraints that are specified in the component meta model should be guaranteed as a consequence of how components are configured. For example, deadlock-freeness, real-time service guarantees, security guarantees, or even maximum transaction cost would be properties that could be ensured at the level of a component architecture and the composition abstractions provided.
- A *composition monitor* will visualize the Piccola agents, channels and forms that realize a composition, and permit users to view and interact with an evolving component configuration.

2.2.4. Work plan

The activities in this project will be iterative and interleaved. Nevertheless, we expect to concentrate on the following topics in the first and second year:

First year

Component meta model:

- Extend FAMIX to express architectural constraints
- Implement a repository with XML import and export facilities
- Develop metrics rendering tool for software evolution

Component migration:

- Software analysis for duplicated code and functionality
- Migration assistants for duplicated functionality
- Language-independent refactorings based on Famix
- Reengineering pattern catalogue

Compositional infrastructure:

- Middleware bridge for Piccola
- Distributed form space
- Composition abstractions

Second year

Component meta model:

- Develop query and navigation facilities for component repository

Component migration:

- Software analysis for architectural drift based on metrics suite
- Migration assistants for architectural drift
- Language-specific refactorings for Java, Smalltalk and C++
- Reengineering pattern catalogue refined to a systematic “pattern language”

Compositional infrastructure:

- Composition monitor

2.2.5. What is the importance of the proposed work?

There is enormous pressure in industry to move software development towards component-based platforms in order to be able to respond more quickly to rapidly changing requirements. Unfortunately the level of maturity in component technology is quite low, except in a few domains which have received a lot of attention from vendors (such as those targeted by 4GL environments). Similarly, component platforms and standards focus more on syntax than semantics, and sidestep many of the more difficult interoperability issues. Finally, there does not exist to this date a culture of investment in software reengineering as a day-to-day part of the software process. Industry as a whole is not prepared to migrate existing software systems to component platforms.

This project proposes to undertake basic research in component migration in an attempt to address some of these shortcomings. Some of the expected results (mainly the software analysis tools and the best practice patterns) are expected to be of industrial relevance in the short term. Other parts of the project (such as component meta model extensions and composition infrastructure) are more foundational in nature, and are expected to have industrial impact only in the very long term.

2.3. International Collaboration

2.3.1. Does this project have an international aspect?

Not formally.

2.3.2. If so, in what form?

We have a collaboration agreement with ABB (Baden and Heidelberg) on the topic of component migration, and we expect some crossover of results with this NFS project.

2.3.3. In which countries do the most important partners operate?

ABB is a worldwide conglomerate. We will interact mainly with the research divisions in Switzerland and Germany.