Proposal form

Project Funding : individual projects (Divisions I-III)
Deadlines : March 1 and October 1

Part 2 : Scientific Information

Main applicant: Nierstrasz, Oscar
Project title: *Tools and Techniques for Decomposing and Composing Software*

Contents

1 Summary

Despite advances in programming languages, software development environments, documentation standards, and software processes, software continues to be hard to develop, hard to understand, and hard to maintain. In particular, no matter how much effort is put into developing clean, modern, software systems, it seems that successful software inevitably drifts towards increasingly complex and hard-to-maintain "legacy systems."

This project proposes to develop new tools and techniques for *decomposing* software systems, that is, for breaking down and understanding complex software, and for *composing* software systems, that is, structuring software so that it becomes easier to maintain, reconfigure, and extend. The proposed work builds on our previous work on the MOOSE reverse engineering environment and the Piccola composition language.

Decomposition: We propose to develop techniques for extracting architectural artifacts from software by (i) applying rule-based reasoning to analyze the software information space, and (ii) providing mechanisms to visualize, reason about and interact with the run-time structures.

- *Code Analysis:* We propose to extend MOOSE with a rule-based interface based on the SOUL logic programming framework. This will allow us to more easily perform various analyses on software systems, such as type reconstruction, evaluation of constraints, and recovery of architectural artifacts. We also propose to develop tools and techniques to detect potential software components, and to classify and group software elements with a view towards component-based reengineering.
- *Run-time Interaction:* We have previously focused on program understanding by means of visualizing simple metrics extracted from complex software systems. We now propose to extend this approach to (i) visualizing artifacts of a running systems, and (ii) interacting with a running system to understand the dynamic architecture.

Composition: We further propose to develop techniques to support high-level composition of applications from software components.

- *Composition Languages:* We plan to develop a successor to Piccola that simplifies the design of compositional styles by making components first-class entities. Reasoning about composition will be supported by a type system that expresses constraints over interfaces of components and the scripts that configure them
- *Compositional Styles:* Existing component models are mostly general-purpose and heavyweight. We propose to develop various lightweight domain-specific component models, or *compositional styles* that reflect the compositional characteristics and constraints of each domain.
- *Composition Mechanisms:* Existing programming languages are better suited for "wiring" components than for *plugging* them together. We propose to experiment with higher-level compositional mechanisms for existing programming languages that explicitly support various notion of components.

Keywords: *Reverse engineering, program understanding, program visualization, software architecture, scripting, component-based software development.*

2 Research plan

2.1 State of Research

The proposed research spans several fields of computer science. We motivate our work in terms of the chronic problems of *software evolution*. The tools and techniques we investigate in this project are related to *program understanding and visualization*, *software architecture*, and *scripting*.

Background: software evolution

Even successful projects are facing problems of evolution or *software aging* [43, 10]. The persistent character of the problems in software development led Pressman to coin the phrase *chronic affliction* as a more apt way to describe the "software crisis" [46]. Most of the effort spent in developing and maintaining a system is devoted to supporting its evolution [52].

Software maintenance is the name given to the process of changing a system after it has been delivered. Sommerville, referring to studies conducted in the eighties [29, 34], states that large organizations devoted at least 50% of their total development effort to maintaining existing systems [52]. McKee in [34] suggests that maintenance effort is between 65% and 75% of the total effort. So maintenance remains the most expensive software development activity. However, the term maintenance is misleading because it gives the impression that this process is just dealing with bug fixes.

A finer analysis of software maintenance shows that software maintenance is often equivalent to forward engineering and not only limited to corrective maintenance [29, 39]. Maintenance activities have been categorized into three different types as follows (the percentage shows the relative effort compared with the total maintenance effort) [52]:

- *Corrective maintenance* (17%) is concerned with fixing reported errors in the software,
- *Adaptive maintenance* (18%) is concerned with adapting the software to a new environment (*e.g.*, platform or OS), and
- *Perfective maintenance* (65%) is concerned with implementing new functional or non-functional requirements.

Chapin et al. [11] further refine these categories to take different forms of software evolution into account.

Clearly most software development "maintenance" is about supporting the evolution of software. Among the reasons that lead to software decay, the most important ones are linked with the dynamics of software itself. Lehman and Belady derived from empirical observations a set of software evolution Laws [28, 27] of which the following two are especially relevant:

> **Continuous Changes.** *"an E-type program*[1] *that is used must be continually adapted else it becomes progressively less satisfactory"* [27]
>
> **Increasing Complexity.** *"As a program is evolved its complexity increases unless work is done to maintain or reduce it."*[27]

To support the continued evolution of legacy software, reengineering techniques must be applied. Chikosky and Cross define reengineering as *"the examination and the alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form."*[12].

[1] E-Type program: a software system that solves a problem or implements a computer application in the real world.

Program understanding and visualization

Among the various approaches to support reverse engineering that have been proposed in the literature, graphical representations of software have long been accepted as comprehension aids.

Many tools make use of static information to visualize software like Rigi [37], Hy+ [13], SeeSoft [15], ShrimpViews [54], TANGO [53] as well as commercial tools like Imagix[2] to name but a few of the more prominent examples. Most publications and tools that address the problem of large-scale static software visualization treat classes as the smallest unit in their visualizations. There are some tools, for instance the FIELD programming environment [48] which have visualized the internals of classes.

Substantial research has also been conducted on runtime information visualization. Various tools and approaches make use of dynamic (trace-based) information such as Program Explorer [26], Jinsight and its ancestors [44, 45], Graphtrace [25]. Various approaches have been discussed like in [24] or [21] where interactions in program executions are being visualized.

Tools for visualizing runtime information typically produce only static pictures, and do not allow the user to interact with the resulting visualizations. Furthermore, most approaches do not scale very well. Indeed, since these tools base themselves on the recording of traces of running programs, the noise which is generated during a trace is hard to filter out. As a consequence of these limitations, it can be hard to assign consistent and meaningful interpretations to the visualizations these tools generate.

The following open questions characterize the current state-of-the-art in program understanding and visualization:

- How can program visualization tools support programmers in a typical software engineering lifecycle?
- What benefit can be obtained by combining dynamic and static information?
- How can animations be used to visualize run-time behaviour?

Finally, a major issue in program visualization is a missing formal foundation for these techniques. Although certain issues, such as assigning a precise interpretation to generated visualizations, would benefit from a formal foundation, other issues seem to fall outside the scope of formalism. As Brown and Hershberger in [22] put it, “Creating effective visualizations of computer programs is an art, not a science.”

Software architecture

Although the vision of mass-produced software components has a long history [33], the realization that standard architectures should drive component-based software development is a relatively recent phenomenon.

Shaw and Garlan survey the state-of-the-art in their classic 1996 book [50], which defines software architectures in terms of *components*, *connectors* and *rules* governing their composition. So-called *architectural description languages* (ADLs), such as Wright [2], Rapide [30], and ACME [19] allow users to specify software architectures and perform certain basic forms of analysis (such as deadlock detection).

A key notion is that of an *architectural style* [1], which captures the common properties of a class of similar architectures (such a *layered* architectures or *dataflow* architectures). Styles may sometimes be combined, but a particular problem is that of *architectural mismatch* [18] which may occur when attempting to use software components in an inappropriate context.

A very different approach to formalizing architectural styles is the medium of *patterns*. The classic book, *Design Patterns* [16], expresses a set of design artifacts in a cookbook style of presentation. Design patterns can be considered to be very fine-grained architectures. Buschmann and his colleagues at Siemens AG have used the same approach to describe various well-known architectural styles [8].

[2] http://www.imagix.com

The work of Murphy [38] introduces "software reflexion models" that show where an engineer's high-level model of the software does and does not agree with a source model, based on a declarative mapping between the two models. Module Interconnection Languages (MILs) [47] can be used to formally describe the global structure of a software system, by specifying the interfaces and interconnections among the components ("modules") that make up the system. These formal descriptions can be processed automatically to verify system integrity. The work of Kim Mens is related to this, but the relations are expressed on a higher level of abstraction using a logic programming language [35].

Whereas the techniques described above depend on *a priori* identification of architectural artifacts, other *a posteriori* techniques can be used to uncover patterns that are implicit in a system. Formal Concept Analysis [17] uses the key notion of identifying similarities among a set of objects based on their properties and showing the obtained relationships in a lattice. In the context of component mining techniques, Concept Analysis is considered to be a promising technique for identifying modules in legacy code. This approach focuses on obtaining alternative proposals to migrate procedural applications to object-oriented ones (specifically from C to C++) [51]. Tonella and Antoniol [55] propose the detection of instances of Design Patterns based on the idea that they can be characterized as a group of classes sharing mutual relations. Godin and Mili [20] focus more on software reengineering and present a framework for dealing with the design and maintenance of class hierarchies.

Despite the growing interest in Software Architecture in recent years, the following open issues remain:

- There currently exists no mainstream ADL.
- It remains hard to extract the architecture of a software system from either the source code or from the runtime structures generated.
- Although Concept Analysis has been used to identify potential modules and classes in procedural code, it has not yet been exploited for identifying components in object-oriented code.

Scripting

Scripting languages are high-level languages for gluing together services implemented in some other host language [5, 42]. Most scripting languages, like perl [56], Python [31], Ruby [14], SmallScript[3] and TCL [41], provide a combination of general-purpose programming constructs, and some domain-specific features to support, for example, text manipulation, system administration, or graphical user interface construction. These languages tend to be dynamically typed, and dynamically compiled, thus supporting rapid application development.

In practice, scripting languages are commonly used to "wire" together services in a low-level, procedural way. In contrast, the goal of a *composition language* is to plug together software components and services in a high-level way, according to a particular architectural style. Various experimental composition languages have been developed in recent years, such as CLAM [49], which focusses on composition of large-scale "megamodules", BML [57], a so-called "Bean Markup Language" for specifying Java Beans configurations in XML, and CoML [7], another XML-based language for specifying configurations of software components.

Coordination languages constitute another class of high-level composition languages that focus on coordinating dependencies between concurrent and distributed tasks. Linda [9] is the archetypical example of such a language, providing nothing more than mechanisms for parallel processes to coordinate their activities by exchanging messages in a shared "tuple space". Many coordination languages adopt a similar approach.

Other approaches are based on a semantic foundation of the Chemical Abstract Machine (CHAM) [6], such as Gamma [4], on the π calculus [36], such as Darwin [32], or on dataflow models, such as Manifold [3].

Finally, certain researchers have applied functional languages such as Haskell to script third party components [23], and others have been investigation concurrent extensions of functional languages, such as functional nets [40] as a means to express software composition.

[3] http://www.smallscript.net/

Although scripting languages have become increasingly popular in the past dozen years, the following issues remain unresolved:

- Scripting languages tend to be either very domain-specific, or just very simple, general-purpose languages. It is hard to tailor them to multiple domains.
- There is a large gap between the formal approach of ADLs, and the useful, but informal approach of scripting languages.
- There is currently no mainstream, high-level language that offers *components* as a first-class programming concept.
- There is no common agreement what mechanisms a high-level language should provide to support component-based software development.

References

[1] Gregory Abowd, Robert Allen, and David Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4):319–364, October 1995.

[2] Robert Allen and David Garlan. The wright architectural specification language. Cmu-cs-96-tb, School of Computer Science, Carnegie Mellon University, Pittsburgh, September 1996.

[3] Farhad Arbab. The IWIM model for coordination of concurrent activites. In Paolo Ciancarini and Chris Hankin, editors, *Proceedings of COORDINATION'96*, volume 1061 of *LNCS*, pages 34–55, Cesena, Italy, 1996. Springer-Verlag.

[4] Jean-Pierre Banâtre and Daniel Le Métayer. The gamma model and its discipline of programming. *Science of Computer programming*, 15:55–77, 1990.

[5] David Barron. *The World of Scripting Languages*. Wiley, December 1999.

[6] Gérard Berry and Gérard Boudol. The chemical abstract machine. In *Proceedings POPL '90*, pages 81–94, San Francisco, Jan 17-19 1990.

[7] Dietrich Birngruber. Coml: Yet another, but simple component composition language. In *Workshop on Composition Languages, WCL'01*, pages 1–13, Vienna, Austria, September 2001.

[8] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stad. *Pattern-Oriented Software Architecture – A System of Patterns*. Wiley, 1996.

[9] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs: a First Course*. MIT Press, cop. 1990, Cambridge, 1990.

[10] Eduardo Casais. Re-engineering object-oriented legacy systems. *Journal of Object-Oriented Programming*, 10(8):45–52, January 1998.

[11] Ned Chapin, Joanne E. Hale, Khaled Md. Khan, Juan F. Rami l, and Wui-Gee Than. Types of software evolution and software maintenance. *Journal of software maintenance and evolution*, 13:3–30, 2001.

[12] Elliot J. Chikofsky and James H. Cross, II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, pages 13–17, January 1990.

[13] M. Consens and A. Mendelzon. Hy+: A hygraph-based query and visualisation system. In *Proceeding of the 1993 ACM SIGMOD International Conference on Management Data, SIGMOD Record Volume 22, No. 2*, pages 511–516, 1993.

[14] Andrew Hunt David Thomas. *Programming Ruby*. Addison Wesley, 2001.

[15] Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner Jr. SeeSoft—A Tool for Visualizing Line Oriented Software Statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, November 1992.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, Mass., 1995.

[17] B. Ganter and R. Wille. *Formal Concept Analysis*. Springer Verlag, 1995.

[18] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, November 1995.

[19] David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 3, pages 47–67. Cambridge University Press, New York, NY, 2000.

[20] Robert Godin and Hafedh Mili. Building and maintaining analysis-level class hierarchies using galois lattices. In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, pages 394–410, October 1993. Published as Proceedings OOPSLA '93, ACM SIGPLAN Notices, volume 28, number 10.

[21] Dean J. Jerding, John T. Stansko, and Thomas Ball. Visualizing interactions in program executions. In *Proceedings of ICSE'97*, pages 360–370, 1997.

[22] Marc H. Brown John Stasko, John Domingue and Blaine A. Price, editors. *Software Visualization - Programming as a Multimedia Experience*. The MIT Press, 1998.

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

[24] R. Kazman and M. Burth. Assessing architectural complexity. Technical report, University of Waterloo, 1995.

[25] Michael F. Kleyn and Paul C. Gingrich. Graphtrace – understanding object-oriented systems using concurrently animated views. In *Proceedings OOPSLA '88, ACM SIGPLAN Notices*, pages 191–205, November 1988. Published as Proceedings OOPSLA '88, ACM SIGPLAN Notices, volume 23, number 11.

[26] Danny B. Lange and Yuichi Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Proceedings of OOPSLA'95*, pages 342–357. ACM Press, 1995.

[27] M. M. Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124, 1996.

[28] M. M. Lehman and L. Belady. *Program Evolution - Processes of Software Change*. London Academic Press, 1985.

[29] B. P. Lientz and E. B. Sawson. *Software Maintenance Management*. Addison-Wesley, 1980.

[30] David C. Luckham, John L. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.

[31] Mark Lutz. *Programming Python*. O'Reilly & Associates, Inc., 1996.

[32] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeffrey Kramer. Specifying distributed software architectures. In *Proceedings ESEC '95*, volume 989 of *LNCS*, pages 137–153. Springer-Verlag, September 1995.

[33] M.D. McIlroy. Mass produced software components. In P. Naur and B. Randell, editors, *Software Engineering*, pages 138–150. NATO Science Committee, January 1969.

[34] J. R. McKee. Maintenance as a function of design. In *Proceedings of AFIPS National Computer Conference*, pages 187–193, 1984.

[35] Kim Mens. *Automating Architectural Conformance Checking by means of Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, 2000.

[36] Robin Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, 1999.

[37] H.A. Müller. *Rigi - A Model for Software System Construction, Integration, and Evaluation based on Module Interface Specifications*. PhD thesis, Rice University, 1986.

[38] G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of SIGSOFT'95, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28. ACM Press, 1995.

[39] J. T. Nosek and P. Palvia. Software maintenance management: changes in the last decade. *Software Maintenance: Research and Practice*, 2(3):157–174, 1990.

[40] Martin Odersky. Functional nets. In *Proc. European Symposium on Programming*, volume 1782 of *LNCS*, pages 1–25. Springer-Verlag, March 2000.

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

[42] John K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31(3):23–30, March 1998.

[43] David Lorge Parnas. Software aging. In *Proceedings of International Conference on Software Engineering*, 1994.

[44] Wim De Pauw, Richard Helm, Doug Kimelman, and John Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, pages 326–337, October 1993.

[45] Wim De Pauw and Gary Sevitsky. Visualizing reference patterns for solving memory leaks in Java. In R. Guerraoui, editor, *Proceedings ECOOP'99*, volume 1628 of *LNCS*, pages 116–134, Lisbon, Portugal, June 1999. Springer-Verlag.

[46] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 1994.

[47] R. Prieto-Diaz and Neighbors J.M. Module interconnection languages. *The Journal of Systems and Software*, 6(4):307–334, November 1986.

[48] Steven P. Reiss. Interacting with the field environment. *Software - Practice and Experience*, 20:89–115, 1990.

[49] Neal Sample, Dorothea Beringer, Laurence Melloul, and Gio Wiederhold. CLAM: Composition language for autonomous megamodules. In Paolo Ciancarini and Alexander L. Wolf, editors, *Proceedings of Coordination'99*, volume 1594 of *LNCS*, pages 291–306, 1999.

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

[51] Michael Siff and Thomas Reps. Identifying modules via concept analysis. In *Proc. of. the Internation Conference on Software Maintenance*, pages 170–179. IEEE Computer Society Press, 1997.

[52] Ian Sommerville. *Software Engineering*. Addison Wesley, fifth edition, 1996.

[53] John T. Stasko. Tango: A framework and system for algorithm animation. *IEEE Computer*, 23(9):27–39, September 1990.

[54] Margaret-Anne D. Storey and Hausi A. Müller. Manipulating and documenting software structures using shrimp views. In *Proceedings of the 1995 International Conference on Software Maintenance*, 1995.

[55] Paolo Tonella and Giuliano Antoniol. Object oriented design pattern inference. In *Proceedings ICSM '99*, pages 230–238, October 1999.

[56] Larry Wall and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., 2nd edition, 1990.

[57] Sanjiva Weerawarana, Francisco Curbera, Matthew J. Duftler, David A. Epstein, and Joseph Kesselman. Bean markup language: A composition language for JavaBeans components. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS-01)*, pages 173–188, Berkeley, California, February 2001. USENIX Association.

2.2 Research fields

Oscar Nierstrasz

The Software Composition Group was founded in 1994, and has carried out both fundamental and applied research since then on issues related to the development of flexible and reconfigurable software systems.

The group has produced a large number of prototypes, publications and theses (diploma and Ph.D.) on the topic of reengineering (see more below on the contributions of Stéphane Ducasse). A key result of these experiences is the forthcoming book, *Object-Oriented Reengineering Patterns* [8], co-authored with Serge Demeyer and Stéphane Ducasse, which describes a number of recurring solutions that experts apply while reengineering and maintaining object-oriented systems. The principles and techniques described in this book have been observed and validated in a number of industrial projects, and reflect best practice in object-oriented reengineering.

In the context of the current and preceding NFS projects, the group has developed an experimental *composition language* called "Piccola" [4, 3]. Piccola is designed as a high-level language for expressing applications as compositions of software components. With Piccola, one describes both compositional (or "architectural") styles for a particular problem domain, and scripts that compose components in that domain. We have developed experimental styles for various domains [2, 21] in the iterative design of the language. Considerable effort was invested in the development of a precise semantics for Piccola [1, 17, 28] to enable reasoning about components.

Although Piccola is fast and stable enough [27] to permit larger scale experiments, there are a number of clear shortcomings. The two most important are: (i) Piccola is missing a suitable type system due to the challenges of open systems, and (ii) the end-user language is too close to the underlying process calculus semantics, making it hard for non-experts to model components.

Roel Wuyts

The following aspects of Wuyts' research are of special interest for this proposal:

- *Systems analysis:* As validation for his PhD [32], Wuyts implemented a logic programming language (called "Soul") to reason about the static structure of object-oriented languages. This language was as the core mechanism to synchronize simultaneous changes to design and implementation [31], to express, check, enforce and search for programming patterns [19, 18] and to express software architectures in such a way that they could be checked against the implementation [20]. Besides reasoning directly on the static structure of systems, it was also used to do event analysis from dynamic information, where it was also integrated with the Moose development environment [25].
- *Language Symbiosis:* The logic programming language that was implemented in the context of the Ph.D. research features a novel form of reflection between two languages supporting different paradigms [32]. This language symbiosis allows to transparently use, change and create objects in the logic programming language. The technique used to achieve this promises to be applicable in other contexts as well, such as the composition of components from different languages.
- *Component Models:* SCG is currently participating in an European ESPRIT project (Pecos, BBW 00.0170) where its responsibility is to develop a component model for embedded devices. Because of the context of small embedded systems, this component model needs to focus on non-functional requirements such as timing and memory consumption.

Stéphane Ducasse

The contributions of Ducasse which are relevant to this proposal are in the domain of reengineering object-oriented systems. These are summarized in [13]: The definition of a language independent meta model [9, 10], the imple-

mentation of a reengineering environment [29], the evaluation of software metrics applied to reengineering [5], [7], the definition of a novel approach for reverse engineering large applications [6, 14], the definition of new approaches for understanding classes [16], language independent detection of duplicated code [15, 26], the use of dynamic information for extracting behavioral views [22, 23, 24, 12, 11], the evaluation of language independent refactorings [30], and the identification of reengineering patterns [8].

References

[1] Franz Achermann. *Forms, Agents and Channels - Defining Composition Abstraction with Style*. PhD thesis, University of Berne, January 2002.

[2] Franz Achermann, Stefan Kneubuehl, and Oscar Nierstrasz. Scripting coordination styles. In António Porto and Gruia-Catalin Roman, editors, *Coordination '2000*, volume 1906 of *LNCS*, pages 19–35, Limassol, Cyprus, September 2000. Springer-Verlag.

[3] Franz Achermann, Markus Lumpe, Jean-Guy Schneider, and Oscar Nierstrasz. Piccola – a small composition language. In Howard Bowman and John Derrick, editors, *Formal Methods for Distributed Processing – A Survey of Object-Oriented Approaches*, pages 403–426. Cambridge University Press, 2001.

[4] Franz Achermann and Oscar Nierstrasz. Applications = Components + Scripts – A Tour of Piccola. In Mehmet Aksit, editor, *Software Architectures and Component Technology*, pages 261–292. Kluwer, 2001.

[5] Serge Demeyer and Stéphane Ducasse. Metrics, do they really help? In Jacques Malenfant, editor, *Proceedings LMO'99 (Languages et Modèles à Objets)*, pages 69–82. HERMES Science Publications, Paris, 1999.

[6] Serge Demeyer, Stéphane Ducasse, and Michele Lanza. A hybrid reverse engineering platform combining metrics and program visualization. In Francoise Balmas, Mike Blaha, and Spencer Rugaber, editors, *Proceedings WCRE'99 (6th Working Conference on Reverse Engineering)*. IEEE, October 1999.

[7] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *Proceedings of OOPSLA'2000, ACM SIGPLAN Notices*, pages 166–178, 2000.

[8] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002. to appear, spring 2002.

[9] Serge Demeyer, Stéphane Ducasse, and Sander Tichelaar. Why unified is not universal. UML shortcomings for coping with round-trip engineering. In Bernhard Rumpe, editor, *Proceedings UML'99 (The Second International Conference on The Unified Modeling Language)*, volume 1723 of *LNCS*, Kaiserslautern, Germany, October 1999. Springer-Verlag.

[10] Serge Demeyer, Sander Tichelaar, and Stéphane Ducasse. FAMIX 2.1 - the FAMOOS information exchange model. Technical report, University of Bern, 2001. to appear.

[11] Stéphane Ducasse. Des techniques de contrôle de l'envoi de messages en smalltalk. *L'Objet*, 3(4):355–377, 1997.

[12] Stéphane Ducasse. Evaluating message passing control techniques in smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, June 1999.

[13] Stéphane Ducasse. Reengineering object-oriented applications. Technical report, Université Pierre et Marie Curie (Paris 6), 2001. Habilitation.

[14] Stéphane Ducasse and Michele Lanza. Towards a methodology for the understanding of object-oriented systems. *Technique et science informatiques*, 20(4):539–566, 2001.

[15] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In Hongji Yang and Lee White, editors, *Proceedings ICSM'99 (International Conference on Software Maintenance)*, pages 109–118. IEEE, September 1999.

[16] Michele Lanza and Stéphane Ducasse. A categorization of classes based on the visualization of their internal structure: the class blueprint. In *Proceedings of OOPSLA 2001*, pages 300–311, 2001.

[17] 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.

[18] K. Mens, I. Michiels, and R. Wuyts. Supporting software development through declaratively codified programming patterns. *SEKE 2001 Special Issue of Elsevier Journal on Expert Systems with Applications*, 2001. To be published; extended version of [19].

[19] Kim Mens, Isabel Michiels, and Roel Wuyts. Supporting software development through declaratively codified programming patterns. In *SEKE 2001 Proceedings*, pages 236–243. Knowledge Systems Institute, 2001. International conference on Software Engineering and Knowledge Engineering, Buenos Aires, Argentina, June 13-15, 2001.

[20] Kim Mens, Roel Wuyts, and Theo D'Hondt. Declaratively codifying software architectures using virtual software classifications. In *Proceedings of TOOLS-Europe 99*, pages 33–45, June 1999.

[21] Oscar Nierstrasz and Franz Achermann. Supporting Compositional Styles for Software Evolution. In *Proceedings International Symposium on Principles of Software Evolution (ISPSE 2000)*, pages 11–19, Kanazawa, Japan, Nov 1-2 2000. IEEE.

[22] Tamar Richner. Describing framework architectures: more than design patterns. In Jan Bosch, Helene Bachatene, Görel Hedin, and Kai Koskimies, editors, *Proceedings of the ECOOP '98 Workshop on Object-Oriented Software Architectures*, Research Report 13/98. University of Karlskrona, July 1998.

[23] Tamar Richner and Stéphane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In Hongji Yang and Lee White, editors, *Proceedings ICSM'99 (International Conference on Software Maintenance)*, pages 13–22. IEEE, September 1999.

[24] Tamar Richner and Stéphane Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. Technical Report IAM-01-007, University of Bern, Institute of Computer Science and Applied Mathematics, December 2001.

[25] Tamar Richner, Stéphane Ducasse, and Roel Wuyts. Understanding object-oriented programs with declarative event analysis. In Serge Demeyer and Jan Bosch, editors, *Object-Oriented Technology (ECOOP'98 Workshop Reader)*, LNCS 1543. Springer-Verlag, July 1998.

[26] Matthias Rieger, Stéphane Ducasse, and Georges Golomingi. Tool support for refactoring duplicated oo code. In *Object-Oriented Technology (ECOOP'99 Workshop Reader)*, number 1743 in LNCS (Lecture Notes in Computer Science). Springer-Verlag, 1999.

[27] Nathanael Schärli. Supporting pure composition by inter-language bridging on the meta-level. Diploma thesis, University of Bern, September 2001.

[28] 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.

[29] Sander Tichelaar, Juan Carlos Cruz, and Serge Demeyer. Design guidelines for coordination components. In Janice Carroll, Ernesto Damiani, Hisham Haddad, and Dave Oppenheim, editors, *Proceedings ACM SAC 2000*, pages 270–277. ACM, March 2000.

[30] Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings ISPSE 2000*, pages 157–167. IEEE, 2000.

[31] Roel Wuyts. Synchronising changes to design and implementation using a declarative meta-programming language. In *International Workshop on (Constraint) Logic Programming for Software Engineering*, dec 2001.

[32] Roel Wuyts and Stéphane Ducasse. Symbiotic reflection between an object-oriented and a logic programming language. In *ECOOP 2001 International workshop on MultiParadigm Programming with Object-Oriented Languages*, 2001.

2.3 Detailed Research Plan

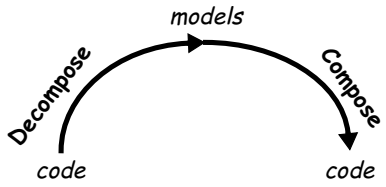
This project builds on results obtained in the ongoing NFS project 20-61655.00, "Meta-models and Tools for Evolution Towards Component Systems", which has succeeded in developing (i) a language-independent meta-model for representing object-oriented software artifacts, (ii) an environment, MOOSE, for storing, analyzing, viewing, and refactoring these artifacts, and (iii) Piccola, a high-level composition language for wrapping existing software as components, and expressing new applications as compositions of components. Although these results are encouraging, there are many open questions. This project proposes to address the following questions:

- How can we recognize architectural artifacts in complex legacy software systems?
- How can we effectively query and navigate through the software information space?
- How can we reconstruct software to make the underlying architecture explicit?

We propose to address these questions by:

1. *Developing prototypes* to explore innovative techniques for analyzing and composing software.
2. *Validating* the techniques by applying them to industrial and open-source case studies.
3. *Disseminating* the most mature tools and techniques.

The research will be carried out by means of two complementary tracks, as illustrated in the graphic below. A *bottom-up* approach will be used to *decompose* software, that is, analyze existing code to develop higher level models in terms of software architecture and component models. A *top-down* approach will be used to *compose* software systems according to high-level models of software components and their corresponding architectural styles.



Decomposition

This work builds on experience with our reverse engineering environment, *MOOSE*, and its related tools. MOOSE functions as a repository for models extracted from software source code, and supports various tools that can present and analyze various views of these models. Although the techniques we have developed are good at getting an overview of the static program structures, they fall short in the following two areas:

- *It is hard to extract and evaluate architectural and design elements.*
- *There is no support for understanding run-time collaborations.*

We therefore propose to explore the following approaches:

- *Architectural analysis:*

  The MOOSE environment currently uses a query-based approach to interrogate a language independent software model. We propose to complement this facility with a rule-based interface by integrating the logic programming language SOUL. By default SOUL reasons about Smalltalk source code and is used to express programming conventions, design pattern structures, software architectures and UML class diagrams. We plan to let SOUL reason about the elements in the MOOSE model instead of directly on Smalltalk code. This integration extends the kinds of metric-based analysis that can currently be done in Moose. In particular, we plan to use this to identify architectural artifacts and check for consistency between the actual software base and design constraints. We will also explore the use of SOUL for type analysis and type reconstruction, and for the evaluation of non-functional constraints (such as timing constraints).

  Whereas logic programming can help us to reveal *anticipated* architectural artifacts in software, Concept Analysis allows us to discover *unanticipated* and hidden patterns in software. We propose to apply Concept Analysis to explore natural groupings of software artifacts that share properties, and to use simple classification mechanisms to explore alternative groupings of elements. These groupings could have different levels of granularity depending on the studied software artifacts (classes, instance variables, methods). Concept Analysis can then be used to detect maximal sets of properties related to a set of entities. This technique provides a way of discovering unknown relationships between software artifacts, based on combinations of simple ones.

- *Run-time Interaction:* MOOSE offers a code-centric infrastructure for reverse engineering. The run-time architecture of a software system cannot, however, be easily extracted from the source code alone. We propose to explore a number of techniques to make these run-time structures more explicit.

  First of all, we propose to apply the metrics-based visualization mechanisms offered by MOOSE and CodeCrawler to run-time structures and to program traces. In this approach, simple metrics will be gathered either at run-time, or extracted from traces, and visualized in various ways. We plan to extend the approach by using grouping mechanisms to better manage the high volume of data. We further plan to combine the dynamic views with the existing static views so that the two can be correlated. Other possible paths of exploration include animated displays and visual or non-visual navigation of run-time information. As with static visualization, the main challenge is to develop a suite of metrics that can not only be easily gathered, but actually provide useful insight into the running behaviour of a system.

  Second of all, we propose to explore extending the paradigm of interactive debugging to higher-level programming constructs, such as groups of collaborating objects. Current debuggers focus on providing detailed, low-level views of program entities to support debugging activities. Instead we would focus on providing high-level views of sets of program entities, running threads, and relationships between them, to enable understanding of the run-time characteristics of design artifacts.

  Finally, we plan to explore the use of run-time information to interactively develop test cases and test suites to document knowledge extracted during reverse engineering, and to facilitate future changes. One of the most useful forms of documentation of a system is a set of test suites that express typical usage scenarios. Test suites are tedious to program by hand, however, as they consist mainly of boilerplate code. A test generation assistant would keep track of the sequence of steps needed to play through a particular scenario, allow the user to identify the interesting states to be tested, and generate the corresponding code for running the same scenario as a test case.

Composition

Piccola demonstrates the feasibility of a high-level composition language providing component-based, compositional interfaces to services provided by a separate, host language. Nevertheless, Piccola is far from providing the ease of use of traditional scripting languages due to the conceptual gap between the mechanisms offered by Piccola and the component-based methodology that it is supposed to support. We plan to address the following issues:

- *It is hard to express and reason about domain-specific compositional styles.*
- *There exists little experience and no guidelines in developing domain-specific styles.*
- *Existing programming languages continue to be of limited usefulness for component-based development.*

We therefore propose the following research activities:

- *Composition Languages:* We plan to develop a successor to Piccola in which components and connectors will be first-class entities. A style will be specified by defining the component interfaces supported by a domain-specific style, and the operators (connectors) that can be used to compose them.

  A key challenge will be to develop a suitable type system that can express both the services *required* by components as well as those that are *provided* in such a way that not only global system knowledge is not required, but also that new components may be introduced and composed at run-time. This is a key requirement for component composition in open systems. A second major challenge is to express and reason about non-functional constraints carried by a compositional style, such as component interaction protocols, real-time constraints, or security restrictions.

  We plan to tackle reasoning about composition by associating logical constraints to component interfaces, and checking constraints at composition time. We will explore the use of SOUL for expressing and checking constraints.

- *Compositional Styles:* Existing component frameworks are general heavyweight in the sense that they support only low-level *wiring* of components rather than high-level *plugging*. We propose to experiment with high-level compositional styles using a variety of platforms, including Piccola.

  In particular, we propose to develop various domain-specific compositional styles that reflect the compositional characteristics and constraints of domains such as embedded systems and heterogeneous web applications.

- *Composition Mechanisms:*

  Existing programming languages offer only limited support for defining compositional abstractions, for plugging components together and for customizing existing components from the outside. We plan to use our experience with Piccola to explore the definition of more suitable mechanisms for existing programming languages. First of all we plan to explore new models of *mixins* to (statically or dynamically) extend existing components from the outside. This would allow composition mechanisms to extend or refine black-box components to suit particular contexts. Second we would combine this approach with language bridging mechanisms based on reflection. This would actually be an extension of the current mechanisms that can be found in both Soul and Piccola. Last but not least we plan to integrate suitable meta-object protocols, so that components can have different interfaces and meta-information to be used by the composition language.

2.4 Timetable

We expect to achieve the following results over the two years of the project:

Year 1

- Integration of SOUL in MOOSE. Experimental use of SOUL to detect and validate architectural and design patterns and constraints.
- Visualization of run-time structures.
- Constraint-based type system for Piccola component model.
- Experimental compositional styles for various domains.
- Experimentation with mixins and other compositional mechanisms for various languages, including Smalltalk and Java.

Year 2

- Applications of concept analysis and classification to software artifacts.
- Dynamic interaction with run-time structures. Assisted generation of test scenarios.
- Composition language with first-class components and connectors.
- Guidelines and techniques for developing and reasoning about compositional styles.
- Suite of language features for component-based software development. Evaluation of mainstream languages and their suitability for CBSD.

2.5 Significance of Research

Despite the adoption of more advanced programming languages, development environments, software development methods, and various process and development standards, industry continues to have difficulty developing and maintaining software. Although software is being developed at a more rapid rate, it is also turning into a complex “legacy” at a faster rate.

This project promises to deliver a variety of techniques to make complex software easier to understand, to maintain and to evolve. The proposed research will be validated by carrying out experiments with industrial and open-source software. Dissemination will be achieved not only by the usual means of publications and talks, but also by making mature prototypes available for external use (as has been successfully done in the past).