2. Scientific part

2.1. Summary and key-words

Modern-day applications are increasingly built from ready-made software components of both fine and coarse granularity. These components typically make architectural assumptions about how communication takes place with other software components. Unfortunately one is often required to integrate components that make different kinds of assumptions, leading to the so-called *architectural mismatch* problem.

Presently there exists a variety of tools, languages and techniques for building systems from components: object-oriented languages, frameworks, 4GLs, scripting languages, user interface builders, middleware, meta-object protocols and so on. There exists also a considerable body of “best practice”, such as design patterns, standard software architectures, and various reflective techniques. Much of this work, however, has been produced in different communities, and it is not clear how, if at all, these techniques can be productively combined in a disciplined way to build heterogeneous software systems.

We propose an approach in which five of these techniques are combined, namely:

- *Black-box frameworks* provide software components that encapsulate useful functionality
- *Scripting languages* allow one to specify compactly and declaratively how software components are plugged together to achieve some desired result
- *Architectural description languages* are used to explicitly specify architectural styles in terms of the interfaces, contracts and composition rules that components must adhere to in order to be composed
- *Glue agents* adapt components that need to bridge architectural styles
- *Coordination models* provide the coordination media that allow distributed glue agents and components to communicate

We expect to achieve results in the following areas:

1. Identify, specify and implement glue and coordination abstractions for composing heterogeneous software components
2. Catalogue and analyse architectural mismatch (glue) problems; experimentally apply our techniques to selected case studies
3. Develop formal models to reason about aspects of type (interface) matching, architectural styles (contracts) and coordination (implementation correctness and optimization)
4. Further develop and refine our experimental languages and tools (Piccola, FLO, CoCo)

2.2. Research plan

2.2.1. State of the art and related work

Frameworks

An object-oriented “framework” is commonly understood as being a collection of abstract and concrete classes implemented in an object-oriented programming language (such as Smalltalk, or C++) which defines a generic, extensible architecture for a software application. One can “instantiate a framework” by defining subclasses of the framework classes that respond to the specific application requirements. Although the technique is very powerful, there are many problems with this kind of framework:

- *Steep learning curve:* an application developer must invest a great deal of time and effort to understand how to use and extend a typical framework.
- *Weak encapsulation:* Extension by inheritance is often referred to as “white-box” reuse, since one must understand the implementation of a component (class) to reuse it effectively.
- *Implicit architecture:* Although a framework defines an architecture, it is hard to find in the source code, since one sees classes and inheritance hierarchies, not interactions between run-time objects.

To address some of these problems, there is increasing interest is so-called “black-box” or component frameworks, which are used and extended primarily by object and class composition rather than by inheritance. In this way an application developer can focus on interfaces rather than implementation details [24].

A lot of expertise concerning frameworks is to be found in the *design pattern* literature, reusable mini-architectures that solve common design problems [12][19]. Reenskaug [42] and Jacobsen [23] describe methods for constructing frameworks.

Scripting

“Scripting languages” are high-level, dynamic languages for putting together software components that are typically programmed in more conventional, compiled languages [37]. A classic example is the Unix, or “Bourne” Shell [11], which allows Unix commands (i.e., C programs) to be plugged together as “pipes and filters”, provided each command in the chain reads its input from the “standard input stream” and sends its output to the “standard output stream.”

Scripting languages have become increasingly popular in recent years as they make it very easy for users (i.e., application developers as well as system administrators and even end-users) to quickly build small, flexible applications from available components. Scripting languages typically support a single, specific architectural style of composing components (analogous to the pipes and filters style supported by the Unix shell), and they typically are designed with a specific application domain in mind (such as system administration, or GUI design).

Some of the more popular and interesting scripting languages include awk [3], Tcl [36], Perl [46], Python [45], AppleScript [8], JavaScript [17]. Each language provides a set of high level language features (e.g., lists, dictionaries, pattern matching and text substitution, events) and an interface to libraries of external components. What is typically missing from these languages is support for multiple architectural styles, and high-level features for coordinating distributed software components (“agents”).

Although much of the work on scripting has been driven by practical needs, and far removed from formal methods, there is nevertheless a large body of work in the area of for-

mal methods that is concerned with compositional software systems (which is the essence of scripting). Among these works, we consider the most important to be (i) the work on process calculi, particularly the π calculus [34] and the languages like PICT [40] that are based on the π calculus (process calculi provide formal models for reasoning about the composition of processes, or software agents); (ii) the work on object calculi, represented by the work of Abadi and Cardelli [1] (which attempts to formalize types systems for reasoning about object composition); (iii) formal models of coordination, many of which are inspired by Boudol's work on the "chemical abstract machine" [10].

Architecture

An emerging discipline within the software engineering community concerns the study of software architectures. This discipline attempts to document, classify and formally specify generic architectural styles and specific software architectures [43][39].

Allen and Garlan have proposed to formally model software architectures using a variety of techniques [2]. With the language Wright, they formalized a decomposition of software elements into components and connectors [6][7]. The composition of the identified components by means of connectors made the system architecture explicit. In a similar way, Darwin [29], a configuration language for distributed systems, supports a description of the architecture by specifying the connections between ports of software agents. Rapide [27] [28] is an "executable architecture definition language" designed for prototyping system architectures. An architecture defined using Rapide consists of interface definitions (corresponding to component specifications), connection rules (like connectors) which describe the connections between the interfaces, and constraints. Rapide is an event-based execution model. Rapid architectures can also be dynamic, in the sense that the number of components in the architecture can vary as can the communication conditions.

Another approach to specifying architectural styles is to document them in the same manner as design patterns. Buschmann et al. take this approach to specify architectural "patterns" such as layered architectures, pipes and filters, and blackboard systems [12].

Finally Garlan et al. [21] describes the "architectural mismatch" problem, i.e. the kind of problems one encounters when integrating systems based on different architectural styles.

Glue

Closely related to scripting is the notion of "glue". Glue code is concerned with "putting things together", but the emphasis is on bridging gaps between architectural styles. A modern example is glue code that wraps a legacy application to work in a networked environment. Although scripting languages are marginally concerned with glue, in the sense that they can be used to glue together components that have not been designed to work together, typically the hard problems are solved at the stage in which the interface is defined between the components and the scripting language, and not in the scripts themselves.

Glue code is often written in languages like Smalltalk (which is good for wrapping legacy code) or C (which is good for gaining access to low-level interfaces). Some general classes of glue problems (like gluing databases, application code and user interfaces) are to a large extent addressed by so-called Fourth generation systems, like Delphi [26].

Another approach to glue as a way of bridging architectural styles, is to apply a simple form of behavioural reflection to intercept and manipulate communications between software components. Reflective capabilities of languages like CLOS [44][25], Smalltalk [22] or OpenC++ [14], have among others been used to introduce persistent objects [38], fault-tolerance [20], security [16], proxy objects [32] [9] and distributed objects [31]. In this context, ACT [5] an extension of SinaST [4] is based on message passing control to connect

components. All of these experiments have shown that reflection can indeed be used to link components based on different architectural styles.

Although there is a certain body of knowledge and "best practice" concerning glue code, there exists no survey or taxonomy of approaches, no catalogue of "glue abstractions", and no language, environment or tools that are well-suited to deal with general glue problems.

Coordination

Coordination languages are languages for coordination concurrent or distributed software agents. They can be seen as "scripting languages for distributed systems."

The prototypical coordination language is Linda [13], which is not really a language at all, but a small set of primitives to allow software agents to communicate with each other through a "coordination medium" known as a "tuple space." These primitives can then be added to any programming language, like Pascal, C or Java, in which the agents are programmed. There is a large number of experimental coordination languages, many of which are based on Linda-like models of coordination [15].

A somewhat different view of coordination is proposed by Malone [30], who considers coordination as the key to understanding dependencies in domains such as computer supported cooperative work and workflow systems. To our mind these views are complementary rather than competing, and we see a key application of coordination models and languages being in application domains such as workflow systems.

Other approaches like Gluons [41], policies [35], and synchronizers [18] allow the definition of explicit entities responsible for the synchronization and coordination of components. Generic Synchronization Policies [33] proposes the possibility to specify synchronization separately from components.

[1] Martín Abadi and Luca Cardelli, *A Theory of Objects*, Springer, 1996.

[2] Gregory Abowd, Robert Allen and David Garlan, "Formalizing Style to Understand Descriptions of Software Architecture," *ACM Transactions on Software Engineering and Methodology*, vol. 4, no. 4, Oct. 1995, pp. 319-364.

[3] Alfred V. Aho, B. Kernighan and P. Weinberger, "Awk — A Pattern Scanning and Processing Language," Report, Bell Telephone Laboratories, Sept 1978.

[4] Mehmet Aksit and Anand Tripathi, "Data Abstraction Mechanisms in SINA/ST," *Proceedings OOPSLA '88, ACM SIGPLAN Notices*, vol. 23, no. 11, Nov 1988, pp. 267-275.

[5] Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans and Akinori Yonezawa, "Abstracting Inter-Object Communications Using Composition Filters," draft manuscript, University of Twente, 1993.

[6] Robert Allen and David Garlan, "Formal Connectors," CMU-CS-94-115, Carnegie Mellon University, March 1994.

[7] Robert J. Allen, "A Formal Approach to Software Architecture," Ph.D. thesis , School of Computer Science, Carnegie Mellon University, Pittsburgh, May 1997.

[8] Apple Computer, *AppleScript Language Guide*, Apple Technical Library, Addison-Wesley, 1993.

[9] John K. Bennett, "The Design and Implementation of Distributed Smalltalk," *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, vol. 22, no. 12, Dec. 1987, pp. 318-330.

[10] Gérard Berry and Gérard Boudol, "The Chemical Abstract Machine," *Proceedings POPL '90*, San Francisco, Jan 17-19, 1990, pp. 81-94.

[11] S.R. Bourne, "The UNIX Shell," *Bell System Technical Journal*, vol. 57, no. 6 (part 2), July-August 1978, pp. 1971-1990.

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

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

[14] Shigeru Chiba and Takashi Masuda, "Designing an Extensible Distributed Language with a Meta-Level Architecture," *Proceedings ECOOP'93*, O. Nierstrasz (Ed.), LNCS 707, Springer-Verlag, Kaiserslautern, Germany, July 1993, pp. 483-502.

[15] Paolo Ciancarini, and Chris Hankin (Ed.), "Coordination Languages and Models," *Proceedings of the First International Conference, COORDINATION'96*, LNCS 1061, Springer-Verlag, April 1996.

[16] Jean-Charles Fabre, "Systèmes sûrs de fontionnement, tolérance aux fautes par protocoles à métaobjets.," *L'Objet*, vol. 3, no. 1, 1997, pp. 9-29.

[17] David Flanagan, *JavaScript: The Definitive Guide, Second Edition*, O'Reilly & Associates, January 1997.

[18] Svend Frølund and Gul Agha, "A Language Framework for Multi-Object Coordination," *Proceedings ECOOP'93*, O. Nierstrasz (Ed.), LNCS 707, Springer-Verlag, Kaiserslautern, Germany, July 1993, pp. 346-360.

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

[20] Jeff Garbus, David Salomon and Brian Tretter, *SYBASE DBA: Survival Guide*, Sams Publishing, 1995.

[21] 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.

[22] Adele Goldberg and Dave Robson, *Smalltalk-80: The Language*, Addison-Wesley, 1989, ISBN: 0-201-13688-0.

[23] Ivar Jacobson, Martin Griss and Patrik Jonsson, *Software Reuse*, Addison-Wesley/ACM Press, 1997.

[24] Ralph E. Johnson and William F. Opdyke, "Refactoring and Aggregation," *Object Technologies for Advanced Software, First JSSST International Symposium*, Lecture Notes in Computer Science, vol. 742, Springer-Verlag, Nov. 1993, pp. 264-278.

[25] Gregor Kiczales, Jim des Rivières and Daniel G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991.

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

[27] 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*, vol. 21, no. 4, April 1995, pp. 336-355.

[28] David C. Luckham and James Vera, "An Event-Based Architecture Definition Language," *IEEE Transactions on Software Engineering*, vol. 21, no. 9, September 1995, pp. 717-734.

[29] Jeff Magee, Naranker Dulay and Jeffrey Kramer, *Specifying Distributed Software Architectures*, 1995, to appear, ESEC 1995 .

[30] Thomas W. Malone and Kevin Crowston, "The Interdisciplinary Study of Coordination," *ACM Computing Surveys*, vol. 26, no. 1, March 1994.

[31] Jeff McAffer, "Meta-level Programming with CodA," *Proceedings ECOOP'95*, W. Olthoff (Ed.), LNCS 952, Springer-Verlag, Aarhus, Denmark, August 1995, pp. 190-214.

[32] Paul L. McCullough, "Transparent Forwarding: First Steps," *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, vol. 22, no. 12, Dec. 1987, pp. 331-341.

[33] Ciaran McHale, "Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance," Ph.D. thesis, Department of Computer Science, Trinity College, Dublin, 1994.

[34] Robin Milner, Joachim Parrow and David Walker, "A Calculus of Mobile Processes, Part I/II," *Information and Computation*, vol. 100, 1992, pp. 1-77.

[35] Naftaly Minsky and Victoria Ungureanu, "Regulated Coordination in Open Distributed Systems," *Proceedings COORDINATION'97*, David Garlan & Daniel Le Mètayer (Ed.), LNCS 1282, Springer-Verlag, Berlin, Germany, September 1997, pp. 81-97.

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

[37] John K. Ousterhout, *Scripting: Higher Level Programming for the 21st Century*, May 1997, White Paper.

[38] Andreas Paepcke, "PCLOS: A Flexible Implementation of CLOS Persistence," *Proceedings ECOOP'88*, S. Gjessing and K. Nygaard (Ed.), LNCS 322, Springer-Verlag, Oslo, August 15-17, 1988, pp. 374-389.

[39] Dewayne E. Perry and Alexander L. Wolf, "Foundations for the Study of Software Architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, October 1992, pp. 40-52.

[40] Benjamin C. Pierce and David N. Turner, "Pict: A Programming Language based on the Pi-Calculus," Technical Report, no. CSCI 476, Computer Science Department, Indiana University, March 1997.

[41] Xavier Pintado, "Gluons: a Support for Software Component Cooperation," *Object Technologies for Advanced Software, First JSSST International Symposium*, Lecture Notes in Computer Science, vol. 742, Springer-Verlag, Nov. 1993, pp. 43-60.

[42] Trygve Reenskaug, *Working with Objects: The OOram Software Engineering Method*, Manning Publications, 1996.

[43] Mary Shaw and David Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.
[44] G.L. Steele, *Common Lisp The Language, Second Edition*, Digital Press, 1990, ISBN: 1-55558-049-1.
[45] Guido van Rossum, *Python Reference Manual*, Stichting Mathematisch Centrum, Amsterdam, 1996.
[46] Larry Wall and Randal L. Schwartz, *Programming Perl*, O'Reilly & Associates, Inc., 1990.

2.3. Contributions to the field by the applicants

As mentioned in the introduction, this project proposes to address the problems of heterogeneous component systems by integrating techniques from frameworks, scripting, architectures, glue and coordination. The following sections list the contributions we have made to each of those research areas.

Frameworks

An object-oriented framework is a particularly effective approach towards organizing software components [58]. An object-oriented framework provides a reusable architecture for a family of applications, where variations in the problem domain are tackled by customising components.

So far, our work has concentrated on how one can turn the design of a framework architecture from an art into a science. In [48], we forward three design guidelines that explain how to construct an open framework architecture. And with FACE [56] and Zypher [49], we provide tool support for respecting and documenting the framework architecture.

We have included a copy of [48] and [58] as samples of our contribution to this area.

Scripting

In an ideal component world, composing an application would correspond to the writing of a small script that creates, customises and assembles some components into a predefined architecture. With the book [57] and the article [59], we provide an overview of the present problems and the possible solutions to achieve this ideal situation.

Our work on the *Piccola* composition language (based on the formal foundation of PiL) [54][55][61][62] provides an experimental platform to validate certain solutions. Piccola is based on the principle that all parts of a composition environment are agents that exchange messages over a communication medium. Composing an application corresponds then to (i) collecting existing component agents and (ii) gluing them together with specially developed glue agents.

We have included a copy of [55] as a sample of our contribution to this area.

Architecture

A system architecture describes how components are plugged together using connectors. With our work on *FLO* [50][51][53] we show that it is possible to embed the notion of a connector into an object-oriented language. In this way connectors become an executable part of a concurrent architecture, thus making the software architecture more explicit at implementation level.

We have included a copy of [51] as a sample of our contribution to this area.

Glue

Glue techniques are required to adapt components that do not really fit the system architecture. Today, almost all component glue is based on wrappers that pack the original component into a new one with a suitable interface. If used frequently, this technique gives rise to serious performance problems.

In [61], we present the results of a literature survey on component systems and glue techniques. With our work on message passing control [52], we have gathered the necessary experience to experiment with the reflective capabilities of a programming language to adapt components. The idea is to intercept the messages that are exchanged between the components and to patch them up so that they fit each other’s interface.

We have included a copy of [61] as a sample of our contribution to this area.

Coordination

Distributed and concurrent applications are still among the hardest ones to build. Components are believed to contribute a lot by shrink-wrapping proven solutions for coordinating distributed behaviour into off the shelf components. With the *CoCo* coordination framework, we are developing a platform to experiment with coordination components in the context of the world-wide web.

We have done a literature survey of contracts as a way to describe the protocol expected by a certain component [47]. This contract survey has inspired our work on coordination components as an explicit representation of a contract between components [63]

We have included a copy of [63] as a sample of our contribution to this area.

[47] Juan Carlos Cruz, Stéphane Ducasse and Patrick Steayert, *Evaluating Contracts in the Object-Oriented Paradigm*, Working Paper.

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

[49] Serge Demeyer, Koen De Hondt and Patrick Steayert, *The Zypher Open Hypermedia Framework*, To Appear in ACM Computing Surveys.

[50] Stéphane Ducasse, Mireille Blay-Fornarino and Anne-Marie Pinna, "A Reflective Model for First Class Dependencies," *Proceedings of OOPSLA'95*, ACM, October 1995, pp. 265—280.

[51] Stéphane Ducasse and Tamar Richner, "Executable Connectors: Towards Reusable Design Elements," *Proceedings of ESEC/FSE'97, LNCS 1301*, 1997, pp. 483—500.

[52] Stéphane Ducasse, "Des techniques de contrôle de l'envoi de messages en Smalltalk," *L'Objet*, vol. 3, no. 4, 1997, pp. 355—377.

[53] Manuel Guenter, "Explicit Connectors for Coordination of Active Objects," Master's thesis, University of Berne, 1998.

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

[55] 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.

[56] Theo Dirk Meijler, Serge Demeyer and Robert Engel, "Making Design Patterns Explicit in FACE, a Framework Adaptive Composition Environment," *Proceedings ESEC/FSE '97*, M. Jazayeri and H. Schauer (Ed.), LNCS 1301, Springer-Verlag, September, 1997, pp. 94-110.

[57] Oscar Nierstrasz and Dennis Tsichritzis (Ed.), *Object-Oriented Software Composition*, Prentice Hall, 1995.

[58] 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.

[59] Oscar Nierstrasz and Theo Dirk Meijler, "Research Directions in Software Composition," *ACM Computing Surveys*, vol. 27, no. 2, June 1995, pp. 262-264.

[60] Oscar Nierstrasz, Jean-Guy Schneider and Markus Lumpe, "Formalizing Composable Software Systems — A Research Agenda," *Proceedings 1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems FMOODS'96*, Chapmann and Hall, 1996, pp. 271-282.

[61] Oscar Nierstrasz and Markus Lumpe, "Komponenten, Komponentenframeworks und Gluing," *HMD — Theorie und Praxis der Wirtschaftsinformatik*, no. 197, September 1997, pp. 8-23.

[62] 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.

[63] Sander Tichelaar, Juan Carlos Cruz and Serge Demeyer, "Coordination as a Variability Aspect in Open Distributed Systems," SCG working paper, submitted, University of Bern, 1998.

2.4. Detailed research plan

Because of the inherent heterogeneity of the emerging software component market, a key problem will always be how to integrate components provided by different vendors. This basic research project will investigate how one can apply framework technology to integrate partial solutions provided by "best practice" technology that includes scripting languages, glue techniques, architectural description languages and finally coordination models and languages.

We propose to develop a set of experimental tools and techniques that build on our previous and ongoing work in the project "Infrastructure For Software Component Frameworks" (FNRS project no. 2000-46947.96), and that integrate best practice in the following way:

- *Composition as Scripting*

  Scripting languages are used to create, customise and assemble components into a predefined architecture. In the ideal case, applications are specified as high-level "scripts" that indicate how individual components are put together. At this time there exists no thorough analysis of scripting languages and the features that make them work well in practice. We propose to carry our such an analysis and to use this as a basis for developing a systematic composition approach. The results will be used to influence the design of our own experimental composition language (Piccola).

- *Support for multiple Architectural Styles*

  Scripting languages typically support only a single architectural style, but in practice, complex applications must deal with multiple styles. A composition approach must allow multiple styles to be expressed and used. An architectural style defines a family of systems in terms of a pattern of structural organization. More specifically, an architectural style defines a vocabulary of components and connector types, and a set of constraints on how they can be combined. High-level syntax is essential to making architectural styles explicit. Object-oriented frameworks go a long way to making such styles explicit by creative and consistent use of operator overloading, and explicit formulation of the contracts governing composition. We propose to investigate specification techniques for supporting multiple architectural styles during application composition.

- *Glue code to bridge gaps in Architectural Styles*

  When components do not fit together, we often have to write "glue code" that bridges the gaps. Although there is a wealth of knowledge concerning "best practice" in writing glue code, and techniques like wrappers and behavioural reflection work well to bridge architectural gaps, there exists no "theory of glue code", or even a comprehensive survey of glue techniques in the literature. We propose to study glue problems, to develop a taxonomy of glue abstractions that solve glue problems, and to develop language support for specifying reusable glue abstraction based on behavioural reflection (i.e., based on "interceptors").

- *Coordination abstractions*

  Increasingly, components are distributed, and even "active" (i.e., autonomous agents). In this case, composing applications from distributed components can be seen as a coordination problem (in the sense of coordination languages and models). A composition language can then be seen as a kind of coordination language. We plan to investigate the application of composition technology to coordination problems in the context of the Esprit Working Group COORDINA. We also propose to focus on glue and composition problems that occur in distributed settings, in particular using middleware, such as CORBA.

Expected Results

We expect to obtain research results in the following four areas:

1. *Frameworks*

   We plan to use CoCo and FLO to (i) specify a variety of domain-specific architectural styles as “black-box” component frameworks; (ii) develop reusable glue abstractions for bridging architectural styles; (iii) further develop reusable abstractions for coordinating distributed software components.

   We therefore plan to use a component approach both for domain-specific components as well as for the abstractions that compose, adapt and coordinate domain components.

2. *Applications*

   We plan to drive and to validate our approach by investigating a number of composition problems in various domains using CoCo and FLO. The COORDINA working group is providing a variety of industrial case studies, some of which we will use as testbeds for our approach. We also plan to carry out a detailed survey of architectural mismatch, or “glue problems”.

   Finally, we plan to investigate the application of Piccola and our glue environment to the integration of heterogeneous internet services (i.e., databases, http servers, CORBA servers, etc.) by means of explicit connectors. This experiment should show if, as we expect, the limits of the existing approaches can be removed in a fully distributed system.

3. *Formal Aspects*

   Our previous work focused on modelling of composition mechanisms and abstractions using a formal process calculus in an effort to better understand the interaction between these features. We now propose to use this formalism to facilitate reasoning about software compositions. We seek to (i) develop a flexible type system for Piccola that will support an easy transition between dynamic type checking, type inference and static type checking, to ease the integration of (static) component libraries and (dynamic) scripts; (ii) investigate formalisms for expressing and validating contracts between software components as an integral part of an architectural style; (iii) use the π-calculus foundation to reason about implementation optimizations, such as reducing the number of needed threads to implement glue agents efficiently and correctly.

4. *Languages and Tools*

   We have previously developed a series of small language prototypes based on our process calculus modelling of composition mechanisms. We seek to extend and refine this work in the form of the Piccola composition language, by incorporating the features need to support scripting, glue techniques, multiple architectural styles, and coordination contracts.

   We also plan to develop various experimental tools for a glue environment for Piccola, which will enable one to visualize, monitor and interact with glue agents. We plan to investigate tools to facilitate the interaction between the experimental platforms (Piccola, FLO and CoCo) and external components, such as libraries, databases, GUI systems, operating systems, internet applications and CORBA servers.

2.5. Work plan

First year

- *Frameworks:* develop a set of reusable glue abstractions based on established "best practice" and incorporate those into FLO; extend CoCo to address bridging of diverse architectural style
- *Applications:* apply CoCo and FLO to COORDINA case studies; catalogue "glue problems" and experimentally apply the Piccola approach to architectural mismatch problems
- *Formal Aspects:* develop a type inference system for Piccola based on type systems for the π-calculus and for record-based object calculi; experiment with formalisms for expressing and validating contracts between concurrent software components and apply within CoCo
- *Language and Tools:* develop a graphical interface for monitoring the progress of Piccola "glue agents"; develop a CORBA interface to Piccola, CoCo and FLO; iterate Piccola, CoCo and FLO based on the experience with frameworks and applications

Second year

- *Frameworks:* apply the reusable glue abstractions of FLO to glue distributed components; apply CoCo to specify common architectural styles as component frameworks
- *Applications:* apply Piccola, CoCo and FLO to the integration of heterogeneous internet services
- *Formal Aspects:* use the underlying π-calculus formalism of Piccola to epitomize performance (such as reducing the number of running threads while preserving liveness); investigate extensions of type systems to express architectural contracts
- *Language and Tools:* continue to iterate Piccola, CoCo and FLO; extend the Piccola language and implementation to incorporate the type inference system.

2.5.1. What is the importance of the proposed work?

Present day applications are increasingly required to integrate functionality and features from heterogeneous software and hardware platforms. Although some extent of "best practice" techniques are widely known and used, there is no systematic support for flexibly configuring and adapting software components to work together in a heterogeneous setting. Most so-called "middleware" approaches offer only a limited, low-level infrastructure for integration, but do not solve the software engineering problems. This project promises to taxonomize, systematize and integrate best practice into a framework-based approach for composing heterogeneous software systems.

2.6. International Collaboration

2.6.1. Within which international programs or organisations will this project be carried out?

Part of this proposal is concerned with the application of framework technology to coordination problems. We are participating in an ESPRIT Working Group, called COORDINA (BBW Nr. 96.0335-1), which consists of academic and industrial partners in Europe who are studying and applying coordination models and languages to problems of coordination distributed components and agents.

2.6.2. In which countries do the principle partners in this project reside?

There are no formal research partners in this project.

The COORDINA partners are in Italy, France, the Netherlands, Denmark, Great Britain, Portugal, Germany and Switzerland.