Components, Connectors and Architectural Patterns

Marco Antonio de Castro Barbosa

Departamento de Informática – Universidade do Minho
Campus de Gualtar
4710-057 – Braga – Portugal
marco.antonio@di.uminho.pt

Abstract. Although the term software component has been around for a long time, component–based programming has become a buzzword since mid 1990's. The basic motivation is to replace conventional programming by system's construction by composition and configuration of reusable off–the–shelf units, often regarded as "abstraction with plugs". Like many other developments in Software Engineering, component–based programming became a popular technology long before sound, mathematical foundation has been put forward. This paper reports on the research, which constitute the PhD project of the author, on a formal calculus for reasoning about and transforming component–based architectures.

1 Introduction

An increasing number of computer based systems are based on the cooperation of distributed, heterogeneous component organized into open software architectures that, moreover, can survive in loosely–coupled environments and be easily adapted to changing application requirements. Such in the case, for example, of applications designed to take advantage of the increased computational power provided by massively parallel systems or of the whole business of Internet–based software development. In order to develop such systems in a systematic way, the focus in development method has switched, along the last decade, from functional to structural issues: both data and processes are encapsulated into software units which are connected into large systems resorting, to a number of techniques intended to support reusability and modifiability. This encapsulation principle is essential to both the *object–oriented* and the more recent *component–based* [9] software engineering paradigms. Component–based software development, is an emerging field with promising solutions for dealing with rapidly changing requirements of presents applications.

In fact, although the term *software component* has been around for a long time, *component–based programming* has become a buzzword since mid 1990's (see, e.g., [7], [9]). The basic motivation is to replace conventional programming by system's construction by composition and configuration of reusable off–the–shelf units, often regarded as *"abstractions with plugs"*. In fact all engineering

disciplines rely on standard components to design and build their artifacts – software engineering should not be the exception.

Reasoning about components brings to scene another recurring theme in software engineering – tent of software architecture understood as the systematic study of the overall organization of a system, both at the static and dynamic levels, and the corresponding composition rules.

This paper attempts to review briefly this area of study int order to introduce the work plan for the author's PhD thesis, which affiliates itself in the overall research on the introduction of mathematically sound methods for describing and reasoning about components and software architectures.

2 Software Components and Architectures

2.1 Components

The expression *software component*, is so semantically overload that referring to it is often a risk. A *component* is a "black-box" entity which both provides and requires services, encapsulated through a public interface which may exhibit both static and behavioral information.

More specifically, a *software component* is a self–contained building block for software systems which can be deployed independently of its manufacture, all dependencies on context being captured by a suitable *interface* description [1]. In most cases, due to the emergence and generalization of ubiquitous computing and mobile applications, component deployment and composition are performed *dynamically* at run–time.

As it happened before with *object* orientation, *component programming* has grown up to popular technologies before consensual definitions and principles, let alone formal foundations, have been put forward.

The starting point of the work we intend to purpose in this thesis is a coalgebraic modelling theory for components documented in, e.g., ([3],[4], [5]) previously defined at Minho. The underlying component model arises from a widespread paradigm for formally approaching systems, design: the so–called model oriented specification method, of which VDM [15] and Z [13] are well–known representatives. Therefore, software components are regard as state–based abstract machines which persist and evolve in time, according to a behavioural model capturing, for example, partiality or (different degrees of) non determinism. Technically, they are regarded as concrete coalgebras for some Set endofunctors, with specified initial, conditions. Coalgebra theory (see e.g., [6],[8] and proceedings of the *Coalgebraic Methods in Computer Science* workshop series, starting in 1998) emphasis behavioural or observational aspects, while making available, in a dual formulation, the definition and proof principles underlying (functional) programming calculi [2].

[1] According to [20], components represent *the primary computational elements and data stores of a system*, typical examples being *servers, filters, objects, blackboards and databases*, and may have *multiple interfaces* each of them defining a *point of interaction between a component and its environment.*

2.2 Software Architecture

Software design received a great deal of attention by researchers in the 1970s. This research arose in response to the unique problems of developing large–scale software systems first recognized in the 1960s [21]. The premise of the research was that design is an activity separate from implementation, requiring special notations, techniques, and tools.

Over the past decade software architecture has received increasing attention as an important subfield of software engineering. During that time there has been considerable progress in developing the technological and methodological basis for treating architectural design as an engineering discipline [28].

The architecture of a software system defines systems in terms of components and interactions among their components [10]. In addition to specifying the structure and topology of the system, the architecture shows the intended correspondence between the system requirements and elements of the constructed system. It can additionally address system–level properties such as capacity, throughput, consistency, and component compatibility. Architectural models clarify structural and semantic differences among components and interactions. Architectural definitions can be composed to define larger systems. Elements are defined independently so they can be re–used in different contexts. The architecture establishes specifications for individual elements to be written in a conventional programming language.

Architectural modelling has been often presented in terms of the so–called architectural styles [1]. For example, Garlan and Shaw [1] describe several common styles for architectures:

- *Pipes and Filters.* This pattern is suitable for applications that require a defined series of independent computations to be performed on ordered data. It is particularly useful when each of the computation can be performed incrementally on a data stream.
- *Data Abstraction and Object-Oriented Organization.* This pattern is suitable for applications in which a central issue is identifying and protecting related bodies of information, especially representation information. When the solution is decomposed to match the natural structure of the data in the problem domain, the components of the solution can encapsulate the data, the essential operations on the data and the integrity constraints, or *invariants*, of the data and operations.
- *Layered Systems.* This pattern is suitable for applications that involve distinct classes of services that can be arranged hierarchically. Often there are layers for basis system–level services, for utilities appropriate to many applications, and for specific tasks of the application.
- *Repositories.* This pattern is suitable for applications in which the central issue is establishing, and maintaining a complex central body of information.

An architectural style, defines a family of such systems in terms of a pattern of structural organization. More specifically, an architectural style determines the vocabulary of components and connectors that can be used in instances of

that style, together with a set of constraints on how they can be combined [1]. These can be include topological constraints on architectural descriptions (e.g., no cycles). Other constraints–say, having to do with execution semantics–might also be parte of the style definition.

A sound basis for software architecture promises benefits for both development and maintenance. Design should benefit from improved abstractions, notations, and analysis. Architectural definitions should help to provide good specifications for programming activities. Since the architectural definition also serves as the specification for system building, the specifications and code will be less likely to diverge. Software architecture typically plays a key role as a bridge between requirements and implementation by providing an abstract description of a system, the architecture exposes certain properties, while hiding others.

Besides dealing with the functionality aspects of a design (i.e., the static organization of this interacting components and the services they are supposed to provide), software architecture should also be concerned with aspects referring to component *mobility* and *performance*.

Mobile components are components which can change their structure. Mobile systems now pervade the informational world and the wider world of which it is a part. It is expected that new components can be added to an existing system without undue disturbance to the services which the existing system provides. This is particularly true when it comes to systems deployed on the Web. Such systems present a challenge for contemporary methods of formal description. The π-calculus [11] is a theory of mobile systems that allows to express systems precisely and reason about their behaviour rigorously.

On the other hand, integration of software specification and performance evaluation model is considered an important and useful task from the early stages of the software life cycle (see e.g., [17], [18]). Actually there are various approaches to derive performance models from software architecture specification, pointing out their features in terms of generality, type of performance model and specification languages, implementation and feedback to the software designer.

How can we reason formally about components and architectures taken into account as much as these aspects as possible?

It is exactly at this level – that of providing sound formal foundations for this area – that this thesis intends to contribute. In the following sections we detail the involved approach built around the work of an architectural pattern.

2.3 Architectural Patterns

As mentioned above, software architecture provides a structural description of the run–time organization of a software system in terms of a (dynamically reconfigurable) collection of components, their externally observable behavioural properties and their mutual interactions. Therefore, architectural analysis addresses *run–time behavioural properties* such as "plugging compatibility", potential for deadlocks, migration, service realizations, etc.

Reasoning about software components (within a grey–box view) should be made within an architectural representation model, at the *semantic* level. For each particular software system such a model is implemented in terms of a *component framework*, i.e., a technological infrastructure that manages components and supports interaction [2].

In practice components are not expected to fit perfectly together, as the pieces of a puzzle. Their composition often has to deal with 'interfacing gaps' which are bridged with some sort of *glue* specification. Moreover, since components in a system are highly independent of their deployment context, the responsibility of coordinating ('orchestrating') their behaviours to achieve system's goals, should lie elsewhere.

Glue specifications are provided by *connectors* which represent component interaction patterns. Examples include pipes, action calls, event broadcasts, as well as shared tuple space or client–server protocols. Typically, connectors are implemented as *configuration scripts*, written in a scripting language (e.g., Perl, C#, or even Haskell [12]). At the specification level, connectors are described by a set of *role interfaces*, which define the interaction scheme for each connected component (cf., similarity to Uml *use cases*).

In this thesis we propose to focus our attention on the notion of *architectural pattern* as an encapsulation of a coordination unit. Informally an *architectural pattern* is composed by:

- A *connector* (eventually a composition of connectors).
- A specification of requirements on the *interfaces* of participant components. In some cases, for example, it is enough to have in the interface a syntactic action signature; in other cases it may be also required to specify that, for example, a collection of actions must be invoked in a particular order.
- An *invariant* property (formulated in a suitable logic) describing extra constraints on the pattern. For example the maximum number of components connectable at the same time or some compatibility condition between component interfaces and connectors. For mobile patterns, location constraints may also be specified as invariants.

The expression *pattern* is borrowed from other domains of Software Engineering, notably from Object orientation, where it first arose. Although the application domain is different here, we take the expression in the usual sense of a recurring design problem and well–proven generic scheme for its solution.

We believe that architectural patterns are fundamental organizational descriptions of the common top–level structure observed in a group of software systems. They can be viewed as templates, expressing and specifying structural properties of their subsystems, and the responsibilities and relationships between

[2] Such technologies evolved from protocols whereby independently developed components could be deployed and collaborate (such as Com, DCom, JavaBeans or Corba) into platforms which offers different runtime services to manage component activation, concurrent execution, security, etc., often taking over many of the tasks of conventional operating systems. That is the case of, e.g., .Net, Ejb or Ccm.

them. It is considered that the selection of an architectural pattern is a fundamental decision during the design of the overall structure of a software system [14].

Patterns share a formal structure. At every level of scale (architectural patterns, design patterns and idioms) they regularly contain the same elements: a name, a short summary, a context, a problem statement (that includes a description of its force), a solution statement (which covers descriptions of its structure structure, participants, basic dynamics and implementation steps), known uses and consequences (describing benefits and liabilities of the pattern).

Architectural patterns are not new in the research on software architecture, although they are usually taken in a sense which is broader than the one we want to deal with int this thesis. In fact, in most cases, architectural patterns are considered in close connection with object–orientation. Object–oriented languages offer language constructs like abstract classes or inheritance, which support the architectural pattern idea in a very elegant way.

Other authors clarify as architectural patterns specification of what D. Garlan and his collaborators call styles. Such specification are actually closer to one notion of pattern as a particular encapsulation of a coordination unit. As an example, consider Arjona and Roberts [16] extension of the *Pipe and Filters* and *Layers* styles to deal with Parallel Programming using these architectures in a more specific way.

Pattern 1:Pipes and filters In that case, each parallel component performs a different step of the computation, following a precise order of operations on ordered data that is passed from one computation stage to another as a flow through the structure.

> **Context** Design a software program for a parallel system, using a certain programming language for certain parallel hardware.
>
> **Problem** The application of a series of ordered but independent computations is required, perhaps as a series of time–step operations, on ordered data. Conceptually, a single data object is transformed. If the computations are carried out serially, the output data set of the first operation would serve as input to the operations during the next step, whose output would in turn serve as input to the subsequent step–operations

Solution Parallelism is represented by the overlapping of operations through time. The operations produce data output that depend on preceding operations on its data input, as incrementally ordered steps over time. Data from different steps are used to generate change of the input over time. The first set of components begins to compute as soon as the first data are available, during the first time–step, following the order of the algorithm. Then, while this computation takes place on the data, the first set of components is free to accept more new data. The results from the second time–step components can also be passed forward to be operated on by a set of components in a third–step, while now the first time–step can accept more new data, and the second time–step operates on the second group of data, and so forth.

Pattern 2: Layers The *Layers* pattern for a parallel systems is an extension of the original Layers Pattern approach [14] with elements of functional parallelism. The order of operations on data is the most important feature. Parallelism is introduced when two or more components of a layer are able to exist simultaneously, performing the same. A components can be created statically, waiting for calls from higher levels, or dynamically, when a call triggers its creation.

Context Design a software pattern for a parallel system, using a certain programming language for certain parallel hardware.

Problem It is necessary to perform a computation repeatedly, that is composed of a series of ordered operations on a set of ordered data. Not every problem meets this creation. consider a program whose output may be the result of just a single complex computation as a series od conceptually ordered simple operations, executed not for value but for effect, at different level os abstraction. An operation at a high level of abstraction requires the evaluation of one or more operations at lower abstraction levels. If this program is carried out serially, it could be viewed as a chain of subroutine calls, evaluated one after another. Generally, performance as execution time is the feature of interest.

Solution Parallelism is introduced by overlapping different level operations through time in layers. When an operation triggers an operation at a higher level, this may also require the evaluation of other operations in lower levels. These operations are triggered by a function call, and may also need the evaluation of operations, as function calls, in even lower levels. Data can be shared, in the form of arguments for the function calls. During the evaluation of operations in each level, usually the higher levels need to wait for a result from lower levels. However, if each layer is represented by more than one component, they can be executed in parallel and service new requests. Therefore, at the same time, several ordered sets of operation can be carried out by the same system. More complex computations may require a larger number of abstraction levels, while simple computations may require a reduced number of levels. The time required to evaluate a computation depends on its complexity or level of abstraction.

3 Research Questions

Starting from the internal definitions of *architectural patterns* given in the previous section, this thesis aims at developing both a formal model for it and an associated transformation calculus. In particular, the basic research questions of this thesis are:

- The identification from the literature and the engineering practice of typical *architectural patterns*. Possible sources of examples are the literature on ADL applications, commercial component frameworks, design methodologies (for example, can *architectural patterns* be extracted from UML *association classes*?), semantic models for component aggregation (e.g., [29]) and connectors (e.g., [25, 26]).
- The definition of a mathematical model for *architectural patterns* based on *coalgebra theory*, building on the emerging interest in the application of coalgebras as foundations for component–based systems. In particular, the connector part may be defined as a coalgebra, the interface requirements as a (partial) behaviour description and the invariant formulated in a modal logic induced by the underlying functor (cf., [27]).
- The study of suitable notions of *pattern equivalence* and *refinement*, the latter understood as *behavioural substitutability* with respect to the properties of interest. In general, a refinement theory studies changes in the representation of a system, entailing a notion of substitution, but not necessarily equivalence. This means that the usage of a system according to its specification is still valid if is actually built according to the (valid) implementation. What is commonly understood by being a valid usage is that the corresponding observable consequences are still the same, or, in a less strict sense, a subset thereof. Refinement of architectural patterns can be addressed at two different levels (at least):

- *The interface level*, which is concerned with what one may call plugging compatibility. The question is whether a component/connector can be transformed, by suitable wiring, to replace another component/connector with a different interface. Note that, once the structure of the interface type encodes the available operations, this may capture situations of extension of component's functionality.
- *The behaviour level*, where the notion of refinement is based on a simulation preorder for the behaviour model specified by strong monad. A number of situations may be captured depending on the simulation preorder adopted and on the refinement relation induced. No determinism reduction is just one possibility among many others.

Although parametric in the behaviour model, the family of component calculi mentioned above is basically equational, i.e., its laws are formulated as bisimulation equalities. By contrast, the general theme of thesis proposed here is the seek for (simulation) inequations, capturing, in component-based design, refinement situations. In broad terms, refinement can be defined as a transformation of an "abstract" into a more "concrete" design, entailing a notion of substitution. Several authors (See, e.g., [19]) have claimed about the relevance of including refinement calculi at all levels of the component-based design research agenda. Being orthogonal to horizontal composition, refinement ought to be considered at both the architectural and scripting (and coordination) levels. The expected 'deliverable' of the thesis is, therefore, a component refinement calculus in the coalgebraic framework just mentioned. This includes both fundamental and applied research in the sense that all theoretical development should be tested against suitable (but real) case studies.

– Development of a calculus of *architectural patterns* composition (both at the *static aggregation* and *dynamic configuration* levels) and refinement, and its application to a number of cases studies.
– Development of a calculus of *architectural patterns* composition (both at the *static aggregation* and *dynamic configuration* levels) and refinement, and its application to a number of cases studies.

The thesis *methodology* should be incremental, in the sense that a hierarchy of semantic models/calculi should be proposed corresponding to different sorts of aspects to be dealt. Typical layers may be:

– *Static* patterns, leading to the definition of the basic model and calculus, building on previous work from ([5], [29]). It should be noted that a theory of behaviour refinement in a coalgebraic setting, even at this elementary level, requires a shift from "naive" semantic universes (such as Set) to order–enriched ones so that a morphism connection does not necessarily collapse into bisimulation. This is, probably, the most demanding aspect of this work, from a mathematical point of view. Preliminary results on this direction are documented in [22].
– *Dynamic* patterns, including a notion of location, component migration and run–time reconfiguration. From a semantics point of view this entails the

need to extend coalgebraic models to deal with mobility — a possibility to be studied is the reframing of coalgebra theory in a *type dependent* setting.
- Addition of *non functional properties*, in particular by assigning in a structural way measures of performance to patterns. From a computer science point of view we would like to see whether refinement can be related, in a clear measurable way, to algorithmic complexity and software architecture performance. On the other hand, the mathematical question here is how to capture the stochastic process algebra models used by, e.g., [23], in a coalgebraic setting.
- *Fuzzy* patterns, capturing uncertainty in component intraconnection. Such patterns may be of interest in the context of web–oriented software architectures. Actually, the web is becoming more and more an *application infrastructure*, rather than a mere repository of information and hypermedia, the purpose for which it was originally devised. This is an entirely new, and rather promising, field for software architecture research. Mathematically, one should look for the coalgebraic formulation of transition systems in which transitions labelled with some sort of fuzzy logic [24] assertion.

Finally, it is desirable that the theoretical work be followed by the (incremental) development of a HASKELL–based platform for *architectural patterns* prototyping. Such a platform will act as a test–bed for concepts developed within the thesis.

References

1. David Garlan and Mary Shaw. An Introduction to Software Architecture. Advances in Software Engineering and Knowledge Engineering. (1993) 1–39
2. R. Backhouse.: An exploration of the Bird-Meertens formalism. CS 8810, Groningen, University.(1988)
3. L. S. Barbosa.: Componentes as processses: An exercise in coalgebraic modeling. In S. F. Smith and C. L. Talcott, editors, FMOODS'2000 - Formal Methods for Open Object-Oriented Distributed Systems. Kluwer Academic Publishers, September (2000) 397–4170
4. L. S. Barbosa.: Coalgebraic structures in program construction. In Proc. of 6th Brazilian Symposium on Programming Languages (invited tutorial), PUC, Rio de Janeiro, June (2002)
5. L. S. Barbosa and J. N. Oliveira.: State-based components made generic. Elect. Notes in Theor. Comp. Sci. (CMCS'03 - Workshop on Coalgebraic Methods in Computer Science). H. Peter Gumm. **82.1** (2003)
6. B. Jacobs and J. Rutten.: A tutorial on (co)algebras and (co)induction. EATCS Bulletin. **62** (1997) 222-259
7. O. Nierstrasz and L. Dami.: Computer-oriennted software technology. in O. Nierstrasz and D. Tsichritzis, editors, Object-Oriented Software Composition. Prentice-Hall International. (1995) 3–28
8. J. Rutten.: Universal coalgebra: A theory of systems. Theor. Comp. Sci. **249(1)** (2000) 3–80 (Revised version of CWI Techn. Rep. CS-R9652 1996).
9. C. Szyperski.: Component Software, Beyond Object-Oriented Programming. Addison-Wesley. (1998)

10. Mary Shaw and Robert DeLine and Daniel V. Klein and Theodore L. Ross and David M. Young and Gregory Zelesnik.: Abstractions for Software Architecture and Tools to Support Them. Software Engineering **21(4)** (1995) 314–335
11. Milner, R.: Communicating and Mobile Processes: the π-Calculus. (1999)
12. Leijen, D. and Meijer, E. and Hook, J.: HASKELL as an Automated Controller. Third International Summer School on Advanced Functional Programming, Braga. Swierstra, S. D. and Henriques, P. R. and Oliveira, J. N. (Eds). Springer Lect. Notes Comp. Sci. (1608), September. (1998) 268–289
13. Spivey, J. M.: The Z Notation: A Reference Manual (2nd ed). Series in Computer Science (1992)
14. M. Shaw.: Patterns for Software Architectures. Pattern Languages of Program Design. Addison-Wesley (1995) 453–462
15. Cliff B. Jones.: Systematic Software Development using VDM. Prentice-Hall. (1990)
16. J. Ortega-Arjona and G. Roberts. Architectural patterns for parallel programming. In Proceedings of the 3rd European Conference on Pattern Languages of Programming and Computing. (1998)
17. Marco Bernardo.: Let's evaluate performance algebraically. ACM Computing Surveys (CSUR). **31(3)** (1999)
18. Domenico Ferrari.: Considerations on the insularity of performance evaluation. IEEE Transactions on Software Engineering. **12(6)** (1986) 678–683
19. Mark Moriconi and Xiaolei Qian.: Correctness and composition of software architectures. Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering (1994) 164–174
20. David Garlan.: Formal Modeling and Analysis of Software Architecture: Components, Connectors, and Events. Proceedings of the 3rd Internation School on Formal Methods for the Design of Computer, Communication and Software Systems: Software Architectures, SFM 2003. **2804** (2003) 1–24
21. Dewayne E. Perry and Alexander L. Wolf.: Foundations for the Study of Software Architecture. ACM SIGSOFT Software Engineering Notes. **17(4)** (1992) 40–52
22. Meng, S. and Barbosa, L.: Refinement of Generic Software Components. UNU/IIST. May (2003)
23. Marco Bernardo and Lorenzo Donatiello and Roberto Gorrieri.: A Formal Approach to the Integration of Performance Aspects in the Modeling and Analysis of Concurrent Systems. Information and Computation. **144(2)** (1998) 83–154
24. Buckley, J. and Eslami, E.: An Introduction to Fuzzy Logic and Fuzzy Sets. Springer Verlag. Physica-Verlag. (2002)
25. Arbab, F.: A channel-based Coordination Model for Component Composition. CWI Tech. Rep. SEN-R0203. CWI, Amsterdam (2002)
26. Arbab, F. and Rutten, J.: A coinductive calculus of component connectors. CWI Tech. Rep. SEN-R0216. CWI, Amsterdam, (To appear in the proceedings of WADT'02) (2002)
27. Kurz, A.: Logics for Coalgebras and Applications to Computer Science. Ph.D. Thesis. Fakultat fur Mathematik, Ludwig-Maximilians Univ., Muenchen. (2001)
28. David Garlan.: Software architecture: a roadmap. ICSE - Future of SE Track (2000) 91–101
29. Barbosa, L. S.: Towards a Calculus of State-based Software Components. Proc. of 7th Brazilian Symposium on Programming Languages (selected papers). (to appear in the *Jour. of Universal Computer Science*), June, Ouro Preto, Brasil (2003)