On Refinement of Software Architectures

Sun Meng[1,3⋆], Luís S. Barbosa[2⋆⋆], and Zhang Naixiao[3]

[1] School of Computing, National University of Singapore, Singapore
[2] Department of Informatics, Minho University, Portugal
[3] LMAM, School of Mathematical Science, Peking University, China
sunm@comp.nus.edu.sg, lsb@di.uminho.pt, znx@pku.edu.cn

Abstract. Although increasingly popular, software component techniques still lack suitable formal foundations on top of which rigorous methodologies for the description and analysis of software architectures could be built. This paper aims to contribute in this direction: building on previous work by the authors on coalgebraic semantics, it discusses component refinement at three different but interrelated levels: *behavioural*, *syntactic*, *i.e.*, relative to component *interfaces*, and *architectural*. Software architectures are defined through component aggregation. On the other hand, such aggregations, no matter how large and complex they are, can also be dealt with as components themselves, which paves the way to a discipline of hierarchical design. In this context, a major contribution of this paper is the introduction of a set of rules for architectural refinement.

Keywords: software component, software architecture, refinement, coalgebra.

1 Introduction

As the size and complexity of software increase continuously, the design and specification of the overall software architecture [28] becomes a central design problem. Software architecture [28] is an important aspect of software engineering, which has a major impact in system's efficiency, adaptability, reusability, and maintainability. Research on software architecture is still in its progressing phase as witnessed by the emergence, in recent years, of a significative number of approaches and methodologies (see, among many other, [3, 10, 14, 22, 27]). In the object-oriented paradigm, where development methods like the Unified Modeling Language (UML) [8, 25, 23] and the Unified Process (UP) [13] are widely used, architectural design forms a critical element of the whole design process [1].

⋆ partially supported by the National Natural Science Foundation of China, under grant 60473056, and a Public Sector Research grant from the Agency of Science, Technology and Research (A*STAR), Singapore.
⋆⋆ funded by the Portuguese Foundation for Science and Technology, in the context of the PURe project, under contract POSI/ICHS/44304/2002

The importance of software architecture for the working software engineer is highlighted by the ubiquitous use of architectural descriptions containing information about systems, subsystems, components and interfaces which comprise the whole architecture. Expressions like 'client-server organization" [7], "layered system", "pipeline", etc., quite popular in the software engineering jargon, denote in fact particular architectural styles.

The primary focus of architecture-driven software development shifted from code organization to the definition and manipulation of coarser-grained architectural elements, their interactions, and the overall interconnection structure. However, there still lacks a systematic approach to the architectural development process encompassing both aggregation and refinement in a coherent way.

Previous work on specification refinement, understood as the the process of transforming an 'abstract' into a more 'concrete' design (see, *e.g.*, Hoare's landmark paper [12]), has concentrated on preservation of invariance properties. There is, however, a wide range of ways of understanding both what substitution means, and what such a transformation should seek for. In *data refinement* [9], for example, the 'concrete' model is required to have *enough redundancy* to represent all the elements of the 'abstract' one. This is captured by the definition of a surjection from the former into the latter (the *retrieve map*). However, these well established refinement approaches [11, 21] are of limited use for refinement of component-based systems, since they are based on semantic frameworks that consider only the relational behaviour of sequential programs.

The main contribution in this paper is a methodology for the refinement of software architectures. Our work is based on a coalgebraic model for component based systems [4–6] in which components can be aggregated through a number of combinators to build hierarchical models of complex systems. Reference [19] introduces the basic results on interface and behavioural refinement of generic components, including a soundness result, upon which a notion of *architectural refinement* is proposed in this paper. Note that, while interface-level refinement is concerned with the manipulation and adjustment of component interfaces, and behavioural refinement relates blackbox behaviours of components, architectural refinement allows us to refine a component by a subsystem architecture as well as to refine a system by another system with a different architecture.

This paper is organized as follows: The underlying coalgebraic model for components and its calculus are briefly reviewed in sections 2 and 3, respectively. Three kinds of refinement relations are introduced in section 4, followed by a family of refinement rules for refinement of system architectures. The paper closes with a brief discussion on what has been achieved in section 5.

2 Components

A software system is defined in terms of a collection of components and connectors among those components. The components interact with each other by the connectors, exchanging information in terms of messages of specified types. Such systems may in turn be used as components in larger designs.

2.1 Components as Coalgebras

We adopt a coalgebraic model for state-based components which follows closely the "*components as coalgebras*" approach proposed by L. Barbosa *et al* in [4, 5]. This approach provides an observational semantics for software components and a generic assembly calculus. Qualificative *generic* means that the proposed constructions are parametric on a (mathematical) model of behaviour.

Components interact with their environment via interfaces. Every interface provides a set of typed channels for receiving and sending messages, acting as a type for the corresponding component. Let $\mathbb{C}$ be a set of channel identifiers. Then a component *interface* is defined as follows: we can define the interface of a component as follows:

Definition 1. *Let $I \subseteq \mathbb{C}$ and $O \subseteq \mathbb{C}$ be sets of typed input and output channels, respectively. The pair (I, O), is called an* interface *and any component p with such an interface is typed as $p : I \to O$.*

In the simplest, deterministic case, the behaviour of a component p is captured by the output it produces, which is determined by the supplied input. But reality is often more complicated, for one may have to deal with components whose behavioural pattern is, *e.g.*, partial or even non deterministic. Therefore, to proceed in a generic way, the behaviour model is abstracted to a strong monad B. Of course, $\mathsf{B} = \mathsf{Id}$ retrieves the simple deterministic behaviour, whereas $\mathsf{B} = \mathcal{P}$ or $\mathsf{B} = \mathsf{Id} + \mathbf{1}$ would model non deterministic or partial behaviour, respectively. Therefore, a component $p : I \to O$ can be modelled by a pointed concrete coalgebra

$$\langle n_p, U_p, \overline{\alpha}_p : U_p \to \mathsf{B}(U_p \times O)^I, u_0 \in U_p \rangle \tag{1}$$

for the **Set** endo-functor $\mathsf{T}^{\mathsf{B}} = \mathsf{B}(\mathsf{Id} \times O)^I$. In detail, n_p is the component's name, a specific value u_0 is taken as its 'initial state' (or 'seed') and the dynamics is captured by currying the state-transition function $\alpha_p : U_p \times I \to \mathsf{B}(U_p \times O)$. Notice that the computation of p will not simply produce an output and a continuation state, but a B-structure of such pairs.

For a component p as given in (1), we define the operators $\mathsf{name}.p$, $\mathsf{in}.p$, $\mathsf{out}.p$ and $\mathsf{beh}.p$ to return n_p, I, O and α_p respectively. In the following sections, for simplicity, we may sometimes omit the occurrence of n_p and u_0, and just use the T^{B}-coalgebra $\langle U_p, \overline{\alpha}_p \rangle$ to denote component p.

Successive observations of a component p reveal its allowed behavioural patterns. For each state value $u \in U_p$, the behaviour of p at u (more precisely, from u onwards) organize itself into a tree-like structure, because it depends on the sequences of input items processed. Such trees, whose arcs are labelled with I values and nodes with O values, can be represented by functions from non empty sequences of I to B-structures of output items. In other words, the space of behaviours of a component with interface (I, O) is the set $(\mathsf{B}O)^{I^+}$, which is in fact the carrier ν_{T} of the final T^{B}-coalgebra $(\nu_{\mathsf{T}}, \omega_{\mathsf{T}} : \nu_{\mathsf{T}} \to \mathsf{T}^{\mathsf{B}}\nu_{\mathsf{T}})$. Therefore, by finality, from any other T^{B}-coalgebra p, there is a unique morphism $[\![\overline{\alpha}_p]\!]$ making

the following diagram to commute:

$$\begin{array}{ccc} \nu_{\mathsf{T}} & \xrightarrow{\omega_{\mathsf{T}}} & \mathsf{B}(\nu_{\mathsf{T}} \times O)^I \\ {\scriptstyle [\![\overline{\alpha}_p]\!]}\uparrow & & \uparrow{\scriptstyle \mathsf{B}([\![\overline{\alpha}_p]\!] \times O)^I} \\ U_p & \xrightarrow{\overline{\alpha}_p} & \mathsf{B}(U_p \times O)^I \end{array}$$

Applying morphism $[\![p]\!]$ to a state value $u \in U_p$ gives the observable behaviour of a sequence of p transitions starting at u. By instantiating B with concrete strong monads, such as $\mathcal{P}$ and $\mathsf{Id} + \mathbf{1}$, it is possible to model different behaviour patterns such as non-determinism and partial behaviour respectively.

3 Architectures

This section recalls the basic mechanisms for component aggregation along the lines of [4–6]. A simple, but precise, notion of software architecture is introduced as a composition pattern for a number of components.

3.1 Composing Components

In the coalgebraic framework revisited in the previous section, components become *arrows* in a (bicategorical) universe $\mathbf{Cp}$ whose objects are sets, which provide types to input/output parameters (the components' *interfaces*), and component morphisms $h : p \longrightarrow q$ are functions relating the state spaces of $p = \langle n_p, U_p, \overline{\alpha}_p : U_p \to \mathsf{B}(U_p \times O)^I, u_p \in U_p \rangle$ and $q = \langle n_q, U_q, \overline{\alpha}_q : U_q \to \mathsf{B}(U_q \times O)^I, u_q \in U_q \rangle$ and satisfying the following *seed preservation* and *homomorphism* conditions:

$$h\, u_p \;=\; u_q \qquad \text{and} \qquad \overline{\alpha}_q \cdot h \;=\; \mathsf{B}\,(h \times O)^I \cdot \overline{\alpha}_p \tag{2}$$

For each triple of objects $\langle I, K, O \rangle$, a composition law is given by functor $;_{I,K,O} : \mathbf{Cp}(I, K) \times \mathbf{Cp}(K, O) \longrightarrow \mathbf{Cp}(I, O)$ whose action on objects p and q is

$$p\,;q \;=\; \langle n_{p;q}, U_p \times U_q, \overline{\alpha}_{p;q}, \langle u_p, u_q \rangle \rangle \qquad \text{with}$$

$$\begin{aligned} \alpha_{p;q} = {} & U_p \times U_q \times I \xrightarrow{\cong} U_p \times I \times U_q \xrightarrow{\alpha_p \times \mathsf{id}} \mathsf{B}(U_p \times K) \times U_q \\ & \xrightarrow{\tau_r} \mathsf{B}(U_p \times K \times U_q) \xrightarrow{\cong} \mathsf{B}(U_p \times (U_q \times K)) \\ & \xrightarrow{\mathsf{B}(\mathsf{id} \times \alpha_q)} \mathsf{B}(U_p \times \mathsf{B}(U_q \times O)) \xrightarrow{\mathsf{B}\tau_l} \mathsf{BB}(U_p \times (U_q \times O)) \\ & \xrightarrow{\cong} \mathsf{BB}(U_p \times U_q \times O) \xrightarrow{\mu} \mathsf{B}(U_p \times U_q \times O) \end{aligned}$$

Similarly, for each object K, an identity law is given by a functor $\mathsf{copy}_K : \mathbf{1} \longrightarrow \mathbf{Cp}(K, K)$ whose action is the constant component $\langle * \in \mathbf{1}, \eta_{\mathbf{1} \times K} \rangle$. Note that the definitions above rely solely on the monadic structure of B.

In [5, 4] a collection of component *combinators* was defined upon **Cp** in a similar parametric way and their properties studied. In particular it was shown that any function $f : A \longrightarrow B$ can be lifted to **Cp** as

$$\ulcorner f \urcorner = \langle n_{\ulcorner f \urcorner}, \mathbf{1}, \eta_{(\mathbf{1} \times B)} \cdot (\mathsf{id} \times f), * \in \mathbf{1} \rangle$$

A *wrapping* mechanism $p[f, g]$ which encodes the pre- and post-composition of a component with **Cp**-lifted functions is defined as a combinator which resembles the *renaming* connective found in process algebras (*e.g.*, [20]). Let $p : I \longrightarrow O$ be a component and consider functions $f : I' \longrightarrow I$ and $g : O \longrightarrow O'$. By $p[f, g]$ we will denote component p wrapped by f and g, typed as $I' \longrightarrow O'$ and defined by input pre-composition with f and output post-composition with g. Formally, the wrapping combinator is a functor

$$-[f, g] : \mathbf{Cp}(I, O) \longrightarrow \mathbf{Cp}(I', O')$$

which is the identity on morphisms and maps a component $\langle n_p, U_p, \overline{\alpha}_p, u_p \rangle$ into $\langle n_{p[f,g]}, U_p, \overline{\alpha}_{p[f,g]}, u_p \rangle$, where

$$\alpha_{p[f,g]} = U_p \times I' \xrightarrow{\mathsf{id} \times f} U_p \times I \xrightarrow{\alpha_p} \mathsf{B}(U_p \times O) \xrightarrow{\mathsf{B}(\mathsf{id} \times g)} \mathsf{B}(U_p \times O')$$

Component aggregation is catered by three generic tensors, capturing, respectively, *external choice* ($\boxplus : I + J \longrightarrow O + R$), *parallel* ($\boxtimes : I \times J \longrightarrow O \times R$) and *concurrent* ($\circledast : I + J + I \times J \longrightarrow O + R + O \times R$) composition. When interacting with $p \boxplus q : I + J \longrightarrow O + R$, the environment chooses either to input a value of type I or one of type J, which triggers the corresponding component (p or q, respectively), producing the relevant output. In its turn, *parallel* composition corresponds to a synchronous product: both components are executed simultaneously when triggered by a pair of legal input values. Note, however, that the behavioural effect, captured by monad B, propagates. For example, if B expresses component failure and one of the arguments fails, the product will fail as well. *Concurrent* composition combines choice and parallel, in the sense that p and q can be executed independently or jointly, depending on the input supplied. Finally, generalized interaction is catered through a sort of 'feedback' mechanism on a subset of the inputs. This can be defined by a new combinator, called *hook*, which connects some input to some output wires and, consequently, forces *part* of the output of a component to be fed back as input. Formally, for each tuple of objects I, O and Z, we define $-\,\hookleftarrow_Z : \mathbf{Cp}(I + Z, O + Z) \longrightarrow \mathbf{Cp}(I + Z, O + Z)$. This combinator is the identity on arrows and maps each component $p : I + Z \longrightarrow O + Z$ to $p \hookleftarrow_Z : I + Z \longrightarrow O + Z$ given by

$$p \hookleftarrow_Z = \langle n_{p \hookleftarrow_Z}, U_p, \overline{\alpha}_{p \hookleftarrow_Z}, u_p \rangle$$

where

$$
\begin{aligned}
\alpha_{p^{\curvearrowleft}_{Z}} \;=\; & U_p \times (I + Z) \xrightarrow{\quad\alpha_p\quad} \mathsf{B}(U_p \times (O + Z)) \\
& \xrightarrow{\mathsf{B}((\mathsf{id}\times\iota_1 + \mathsf{id}\times\iota_2)\cdot \mathsf{dr})} \mathsf{B}(U_p \times (O + Z) + U_p \times (I + Z)) \\
& \xrightarrow{\mathsf{B}(\eta + a_p)} \mathsf{B}(\mathsf{B}(U_p \times (O + Z)) + \mathsf{B}(U_p \times (O + Z))) \\
& \xrightarrow{\mu\cdot\mathsf{B}\nabla} \mathsf{B}(U_p \times (O + Z))
\end{aligned}
$$

3.2 Systems

From the architectural point of view, a software system comprises a finite set of interconnected components. In itself such a system can be thought of as a new component, which paves the way to hierarchical decomposition. this motivates the following definition:

Definition 2. *A system is defined as a tuple* $S = \langle n_S, I_S, O_S, C, R\rangle$*, where* n_S *is its unique identifier,* $I_S \subseteq \mathbb{C}$ *and* $O_S \subseteq \mathbb{C}$ *are the sets of input and output channels, respectively,* $C = \{p_k\}_{1\leq k\leq n}$ *denotes a finite set of components* $p_k = \langle n_k, U_k, \overline{\alpha}_k : U_k \to \mathsf{B}(U_k \times O_k)^{I_k}, u_k \in U_k\rangle$ *for* $k = 1, 2, \cdots, n$*,* $R = \{\langle op_j, C_j\rangle\}_{1\leq j\leq m}$ *denotes a finite set of combinators together with the components being combined by them, where* $C_j = \{p_{j_1}, p_{j_2}, \cdots \mid \forall i. p_{j_i} \in C\}$*.*

Note that it is useful to introduce a notion of (input/output) channels I_S and O_S as system's external interfaces. Therefore, for a given system S, we define the operators $\mathsf{name}.S$, $\mathsf{in}.S$, $\mathsf{out}.S$, $\mathsf{comps}.S$ and $\mathsf{combs}.S$ to return n_S, I_S, O_S, C and R respectively. Moreover, we have

$$\mathsf{in}.C \triangleq \bigcup_{c\in C} \mathsf{in}.c \quad \text{and} \quad \mathsf{out}.C \triangleq \bigcup_{c\in C} \mathsf{out}.c$$

as the union of input or output channels for the components in S.

In fact, we hope to decompose systems hierarchically, and regard them as ordinary component. Therefore, we introduce the set of channels I_S and O_S as the external interfaces of the system.

3.3 Black-Box and Glass-Box Views of Systems

There are two ways of interpreting a system's specification. The first one emphasises its black-box behaviour and arises from the observation that a system, being composed by component aggregation, is itself a component, actually a (final) coalgebra over its space of behaviours. In other words, as a component whose state space is specified as $(\mathsf{B}O_S)^{{I_S}^+}$. Such component abstracts over all internal structure the system may bear, and is simply defined as

$$p_S = \langle n_S, U_S = (\mathsf{B}O_S)^{{I_S}^+}, \overline{\alpha}_S : U_S \to \mathsf{B}(U_S \times O_S)^{I_S}, \langle u_1, u_2, \cdots, u_n\rangle\rangle$$

For a given system S, we use the notation $[\![S]\!]$ to denote p_S which captures only the externally visible behaviour of the system. Thus, its internal architecture and organization is not characterized by such an interpretation. It does not reflect, for example, the internal structural decomposition of the system, the internal communication between its components, its internal states, and so on. Thus, it gives a pure black-box view of the system, which is mainly used in the early stages of a system development.

In later stages of development, the software engineer is also concerned with structural aspects of the design, and a glass-box view is then required. Such a glass-box view is provided by Definition 2, on top of which a hierarchical decomposition function ξ is defined. Formally,

Definition 3. *For a given system $S = \langle n_S, I_S, O_S, C, R\rangle$, a decomposition function $\xi : C \longrightarrow \mathcal{P}(C)$ is a function which satisfies:*

- $\exists\,!\, p \in C \,.\, p \notin \bigcup ran(\xi)$, *denoted by* ξ^S_{root};
- $\bigcup ran(\xi) = C \setminus \{\xi^S_{root}\}$ *and* $\forall p \in C \setminus \{\xi^S_{root}\} \,.\, (\exists\,!\, p' \in C \setminus \{p\} \,.\, p \in \xi(p'))$;
- $\forall C' \subseteq C \,.\, (C' \neq \emptyset \Rightarrow (\exists p \in C' \,.\, C' \cap \xi(p) = \emptyset))$.

4 Architecture Refinement

From a practical point of view, it is impossible to get a concrete architecture of a large system from the abstract requirements in just one step. Therefore, a stepwise development process is needed where software architectures are refined systematically in a number of steps. In this section, we investigate three kinds of refinement relations, namely, *behavioural*, *interface* and *architectural refinement*.

4.1 Behavioural Refinement

The most fundamental notion of refinement in our approach is behavioural refinement [19], based on a simulation preorder between components with identical interfaces. Since morphisms between such components are in fact coalgebra homomorphisms, therefore entailing bisimilarity, there is a need to seek for a weaker notion of a morphism between components, still preserving the source component dynamics.

We say that a component p behaviourally refines component q if the behavioural patterns observed for p are a structural restriction, with respect to the behavioural model captured by monad B, of those of q. To make such a 'definition' more precise we describe behavioural patterns concretely as *generalized transitions*. Thus a possible (and intuitive) way of regarding component p as a behavioural refinement of q is to consider that p transitions are *preserved* in q. For non deterministic components this is understood simply as set inclusion. But one may also want to consider additional restrictions. For example, to stipulate that if p has no transitions from a given state, q should also have no transitions from the corresponding state(s). Recall that a component morphism from p to q is a seed preserving function $h : U_p \longrightarrow U_q$ such that $\mathsf{B}(h \times \mathsf{id})\cdot\alpha_p \;=\; \alpha_q\cdot(h\times\mathsf{id})$.

In terms of transitions, this equation is translated into the following two requirements (by a straightforward generalization of an argument in [26]):

$$u \xrightarrow{\langle i,o\rangle}_p u' \;\Rightarrow\; h\,u \xrightarrow{\langle i,o\rangle}_q h\,u' \tag{3}$$

$$h\,u \xrightarrow{\langle i,o\rangle}_q v' \;\Rightarrow\; \exists_{u'\in U}\, s.t.\, u \xrightarrow{\langle i,o\rangle}_p u' \wedge v' = h\,u' \tag{4}$$

which captures the fact that, not only p dynamics, as represented by the induced transition relation, is *preserved* by h (3), but also q dynamics is *reflected* back over the same h (4).

To define a weaker notion of coalgebra morphism, let $\leq$ be an order on a **Set** endo-functor T [15] (concretely, mapping every set U into a collection of preorders $\leq_{\mathsf{T}U}$), referred to as a *refinement preorder*. Also assume that functor T is stable wrt order $\leq$[4]. Then,

Definition 4. *Let T be an extended polynomial functor on* **Set** *and consider two T-coalgebras $p = (U, \alpha : U \to \mathsf{T}(U))$ and $q = (V, \beta : V \to \mathsf{T}(V))$. A* forward *morphism $h : p \to q$ with respect to a refinement preorder $\leq$, is a function from U to V such that*

$$\mathsf{T}\,h \cdot \alpha \leq \beta \cdot h$$

The existence of a *forward* morphism connecting two components p and q witnesses a refinement situation whose symmetric closure coincides, as expected, with bisimulation. *Behavioural refinement* is therefore defined as the existence of a forward morphism *up to bisimulation* [5]. Formally,

Definition 5. *Given components p and q, p is a* behavioural refinement *of q, written $p \sqsubseteq_B q$, if there exist components r and s such that $p \sim r$, $q \sim s$ and $r \sqsubseteq_F s$, where $r \sqsubseteq_F s$ stands for the existence of a (seed preserving) forward morphism h from r to s.*

We refer to p as the *concrete* or *refined* component and q as the *abstract* component.

In [19] we have proved the soundness of simulation for behavioural refinement, which is given in the following lemma[6]:

Lemma 1. *To prove $p \sqsubseteq_B q$ it is sufficient to exhibit a simulation R relating components p and q.*

[4] Given a **Set** endofunctor T and a refinement preorder $\leq$, a lax relation lifting is an operation $Rel_{\leq}(\mathsf{T})$ mapping relation R to $\leq \circ Rel(\mathsf{T})(R) \circ \leq$, where $Rel(\mathsf{T})(R)$ is the lifting of R to T (defined, as usual, as the T-image of inclusion $\langle r_1, r_2\rangle : R \longrightarrow U \times V$, *i.e.*, $\langle \mathsf{T}r_1, \mathsf{T}r_2\rangle : \mathsf{T}R \longrightarrow \mathsf{T}U \times \mathsf{T}V$). A functor T is stable wrt a order $\leq$ if the associated lax relation lifting operation $Rel_{\leq}(\mathsf{T})$ commutes with substitution.

[5] In [19] the dual notion of a *backwards* morphism, *i.e.*, one that satisfies $\beta \cdot h \leq \mathsf{T}\,h \cdot \alpha$, is also studied, leading to a notion of *backward* refinement which do have some applications, although the underlying intuition seems less familiar.

[6] Here we adopt a generic definition of simulation due to Jacobs and Hughes in [15]: Given T-coalgebras α and β, a simulation is a $Rel_{\leq}(\mathsf{T})$-coalgebra over α and β, *i.e.*, a relation R such that, for all $u \in U, v \in V$, $\langle u, v\rangle \in R \;\Rightarrow\; \langle \alpha\,u, \beta\,v\rangle \in Rel_{\leq}(\mathsf{T})(R)$.

On the other hand, for two components p and q, if p behaviourally refines q, then we can always get a simulation R between them, which is defined as $\sim \circ Graph(h) \circ \sim$. To prove the result, we first recall from [15] the following result:

Lemma 2. *Let* T *be a functor stable wrt order* $\leq$*. Then,*

- *If* R *is a bisimulation, then both* R *and* R^{op} *are simulations;*
- *Simulations are closed under composition.*

and prove that

Lemma 3. *The graph of a forward morphism* h *between two* T*-coalgebras* $p = (U, \alpha)$ *and* $q = (V, \beta)$ *is a simulation.*

Proof. Define a relation $R \subseteq U \times V$ as $\langle u, v \rangle \in R$ iff $h(u) = v$. Because h is a forward morphism, for all $u \in U$, the following diagram commutes:

$$\begin{array}{ccc} u & \xrightarrow{\quad h \quad} & h(u) = v \\ {\scriptstyle\alpha}\downarrow & & \downarrow{\scriptstyle\beta} \\ \alpha(u) & \xrightarrow{\mathsf{T}h} \mathsf{T}h(\alpha(u)) \leq_{\mathsf{T}V} & \beta(v) \end{array}$$

Since $\leq$ is a preorder, we have $\alpha(u) \leq_{\mathsf{T}U} \alpha(u)$. Therefore, for any $u \in U$ and $v \in V$, if $\langle u, v \rangle \in R$, then $\langle \alpha(u), \beta(v) \rangle \in \leq \circ Rel(\mathsf{T})(R) \circ \leq$. That means, R is a simulation.

Then,

Theorem 1. *For two components* p *and* q*, if* p *behaviourally refines* q*, and this is witnessed by a forward morphism* h*, then* $\sim \circ Graph(h) \circ \sim$ *is a simulation between them.*

Proof. Immediate by combination of lemmas 2 and 3.

4.2 Properties of Behavioural Refinement

Behavioural refinement of components has a number of pleasant properties. First of all it is a preorder:

$$p \sqsubseteq_B p$$
$$p \sqsubseteq_B q \wedge q \sqsubseteq_B r \Rightarrow p \sqsubseteq_B r$$

Proof. The reflexivity is obvious: we just need to take the identity function id on p as the forward morphism (the graph of id is a bisimulation). For the transitivity, we can first derive two simulations R and R' from $p \sqsubseteq_B q$ and $q \sqsubseteq_B r$ respectively, then from Lemma 2, we can know that $R' \circ R$ is also a simulation. By Lemma 1, p is a behaviour refinement of r.

In the case of a large system consisting of many components, it is not practical to consider the whole system each time one of its components is to be refined. On the contrary, we would like to decompose the original system, perform refinement locally and reconstruct the relevant services from the refined components. To make this possible behavioural refinement should also be a pre-congruence. Formally,

Lemma 4. *For any refinement preorder $\leq$, behavioural refinement $\sqsubseteq_B$ is monotonic with respect to combinators:*

$$\begin{aligned} p[f,g] &\sqsubseteq_B q[f,g] \\ p \,;\, r &\sqsubseteq_B q \,;\, t \\ p \boxtimes r &\sqsubseteq_B q \boxtimes t \\ p \boxplus r &\sqsubseteq_B q \boxplus t \\ p \circledast r &\sqsubseteq_B q \circledast t \\ p \,{}^{\backprime}_{Z} &\sqsubseteq_B q \,{}^{\backprime}_{Z} \end{aligned}$$

whenever $p \sqsubseteq_B q$ and $r \sqsubseteq_B t$.

Proof. Let R_1 and R_2 be the simulation relations witnessing $p \sqsubseteq_B q$ and $r \sqsubseteq_B t$ respectively. For wrapping and the hook combinator, we just need to define $R = R_1$. Let $\langle u, v\rangle \in R$, then for all $i \in I$, we can easily derive $\langle \alpha_p(u,i), \alpha_q(v,i)\rangle \in \leq \circ Rel_{\leq}(\mathsf{B}(\mathsf{Id} \times O))(R) \circ \leq$ from $p \sqsubseteq_B q$. Therefore,

$$\begin{aligned} & \langle \overline{\alpha}_{p[f,g]}(u), \overline{\alpha}_{q[f,g]}(v)\rangle \in \leq \circ Rel_{\leq}(\mathsf{B}(\mathsf{Id} \times O')^{I'})(R) \circ \leq \\ \equiv & \forall i' \in I'.\, \langle \alpha_{p[f,g]}(u,i'), \alpha_{q[f,g]}(v,i')\rangle \in \leq \circ Rel_{\leq}(\mathsf{B}(\mathsf{Id} \times O'))(R) \circ \leq \\ \equiv & \langle \mathsf{B}(\mathsf{id} \times g) \cdot \alpha_p \cdot (\mathsf{id} \times f)(u,i'), \mathsf{B}(\mathsf{id} \times g) \cdot \alpha_q \cdot (\mathsf{id} \times f)(v,i')\rangle \in \\ & \leq \circ Rel_{\leq}(\mathsf{B}(\mathsf{Id} \times O'))(R) \circ \leq \\ \equiv & \{\text{let } f(i') = i\} \\ & \langle \mathsf{B}(\mathsf{id} \times g) \cdot \alpha_p(u,i), \mathsf{B}(\mathsf{id} \times g) \cdot \alpha_q(v,i)\rangle \in \leq \circ Rel_{\leq}(\mathsf{B}(\mathsf{Id} \times O'))(R) \circ \leq \\ \equiv & \mathsf{B}(\mathsf{id} \times g) \cdot \langle \alpha_p(u,i), \alpha_q(v,i)\rangle \in \leq \circ Rel_{\leq}(\mathsf{B}(\mathsf{Id} \times O'))(R) \circ \leq \\ \equiv & \langle \alpha_p(u,i), \alpha_q(v,i)\rangle \in \leq \circ Rel_{\leq}(\mathsf{B}(\mathsf{Id} \times O))(R) \circ \leq \\ \equiv & \text{TRUE} \end{aligned}$$

The proof for the hook combinator can be similarly obtained. Proofs for the monotonicity of $\sqsubseteq_B$ for other combinators can be found in [18].

4.3 Interface Refinement

Behavioural refinement characterizes the preservation of component behaviour. But if we rely solely on behavioural refinement, the inability to change the syntactic interface will force us to work at the same level of interface abstraction throughout the whole development process. To avoid this, a more general notion of refinement, called interface refinement is introduced, which relates components with different interfaces.

Definition 6. *Let $p : I \to O$ and $q : I' \to O'$ be components. If there exist functions $w_1 : I' \to I$ and $w_2 : O \to O'$, such that*

$$p[w_1, w_2] \sqsubseteq_B q$$

then p is an interface refinement *of q modulo the downwards function w_1 and the upwards function w_2, written as $p \sqsubseteq_{(w_1,w_2)} q$.*

Interface refinement supports the systematic construction of new components from existing ones. Generally, for any component p, and functions w_1, w_2,

$$p \sqsubseteq_{(w_1,w_2)} p[w_1, w_2]$$

One situation where this technique is useful is when we have an already completed off-the-shelf component and want to adapt the syntactic interface of this component to fit some context requirements. Therefore, interface refinement provides a systematic pattern for interface adaptation of components.

As explained previously, the behavioural refinement relation on components is both reflexive and transitive. Moreover, behavioural refinement is monotonic with respect to the combinators defined in the component calculus. This allows system development in a flexible top-down manner. The following properties show that interface refinement combines nicely with behavioural refinement:

$$p_1 \sqsubseteq_B p_2 \wedge p_2 \sqsubseteq_{(w_1,w_2)} p_3 \Rightarrow p_1 \sqsubseteq_{(w_1,w_2))} p_3$$
$$p_1 \sqsubseteq_{(w_1,w_2)} p_2 \wedge p_2 \sqsubseteq_B p_3 \Rightarrow p_1 \sqsubseteq_{(w_1,w_2))} p_3$$

Furthermore, we have transitivity in the sense that

$$p_1 \sqsubseteq_{(w_1,w_2)} p_2 \wedge p_2 \sqsubseteq_{(w_3,w_4)} p_3 \Rightarrow p_1 \sqsubseteq_{(w_3 \cdot w_1, w_2 \cdot w_4)} p_3$$

4.4 Architectural Refinement

By *architectural* refinement we mean behavioural refinement of a complex system regarded as a component on its own. Formally,

Definition 7. *Let $S = \langle n_S, I_S, O_S, C, R\rangle$ and $S' = \langle n_{S'}, I_{S'}, O_{S'}, C', R'\rangle$ be two systems. If $I_S = I_{S'}$, $O_S = O_{S'}$, and $[\![S]\!] \sqsubseteq_B [\![S']\!]$, then we say that S is an architectural refinement of S', written as $S \sqsubseteq_A S'$.*

From the transitivity of behavioural refinement relation, one gets,

$$S_1 \sqsubseteq_A S_2 \wedge S_2 \sqsubseteq_A S_3 \Rightarrow S_1 \sqsubseteq_A S_3$$

This definition becomes really useful if it can be translated on concrete refinement rules concerned with structural changes in the design. A number of them, easily derived from the definitions, are stated below in the format

$$\frac{precondition}{refinement}$$

where *precondition* is the condition to be satisfied for the refinement relation to hold.

Behavioural refinement. A system can be refined by refining one of its components and leaving other components unchanged. For a given system $S = \langle n_S, I_S, O_S, C, R\rangle$, let $p \in C$ be a component and p' is a behavioural refinement of p, then we can get a refinement of the whole system:

$$\frac{\begin{array}{c} p \in C \\ p' \sqsubseteq_B p \\ C' = (C \setminus \{p\}) \cup \{p'\} \\ R' = \{\langle op_j, (C_j \setminus \{p\}) \cup \{p'\} \lhd p \in C_j \rhd C_j\rangle\}_{1\leq j\leq m} \\ S' = \langle n_S, I_S, O_S, C', R'\rangle \end{array}}{S' \sqsubseteq_A S}$$

Adding output channels. New output channels may be added to a component p if it is neither connected to a system component, nor part of the system interface. For a given system $S = \langle n_S, I_S, O_S, C, R\rangle$, let $p \in C$ be a component, then

$$\frac{\begin{array}{c} O' \subseteq \mathbb{C} \setminus (\mathsf{in}.C \cup \mathsf{out}.C) \\ p = \langle n_p, U_p, \overline{\alpha}_p : U_p \to \mathsf{B}(U_p \times O_p)^{I_p}, u_p \in U_p\rangle \\ p' = \langle n_p, U_p, \overline{\alpha}_{p'} : U_p \to \mathsf{B}(U_p \times (O_p + O'))^{I_p}, u_p \in U_p\rangle \\ \forall u \in U_p, i \in I_p \,.\, \alpha_{p'}(u,i) = \alpha_p(u,i) \\ C' = (C \setminus \{p\}) \cup \{p'\} \\ R' = \{\langle op_j, (C_j \setminus \{p\}) \cup \{p'\} \lhd p \in C_j \rhd C_j\rangle\}_{1\leq j\leq m} \\ S' = \langle n_S, I_S, O_S, C', R'\rangle \end{array}}{S' \sqsubseteq_A S}$$

Removing output channels. An output channel of component p being not used in the system can be removed from the component. For a given system $S = \langle n_S, I_S, O_S, C, R\rangle$, let $p \in C$ be a component, then

$$\frac{\begin{array}{c} o \notin O_S \cup \mathsf{in}.C \\ p = \langle n_p, U_p, \overline{\alpha}_p : U_p \to \mathsf{B}(U_p \times O_p)^{I_p}, u_p \in U_p\rangle \\ O'_p = O_p \setminus \{o\} \\ p' = \langle n_p, U_p, \overline{\alpha}_{p'} : U_p \to \mathsf{B}(U_p \times O'_p)^{I_p}, u_p \in U_p\rangle \\ \forall u \in U_p, i \in I_p \,.\, \alpha_{p'}(u,i) = \alpha_p(u,i) \\ C' = (C \setminus \{p\}) \cup \{p'\} \\ R' = \{\langle op_j, (C_j \setminus \{p\}) \cup \{p'\} \lhd p \in C_j \rhd C_j\rangle\}_{1\leq j\leq m} \\ S' = \langle n_S, I_S, O_S, C', R'\rangle \end{array}}{S' \sqsubseteq_A S}$$

Adding input channels. An input channel can be added to a component p provided that it is already in the output of some other component or input of the system. For a given system $S = \langle n_S, I_S, O_S, C, R\rangle$, let $p \in C$ be a component,

then

$$\frac{\begin{array}{c} i' \in I_S \cup \mathsf{out}.C \\ p = \langle n_p, U_p, \overline{\alpha}_p : U_p \to \mathsf{B}(U_p \times O_p)^{I_p}, u_p \in U_p \rangle \\ I'_p = I_p \cup \{i'\} \\ p' = \langle n_p, U_p, \overline{\alpha}_{p'} : U_p \to \mathsf{B}(U_p \times O_p)^{I'_p}, u_p \in U_p \rangle \\ \forall u \in U_p, i \in I_p \,.\, \alpha_{p'}(u,i) = \alpha_p(u,i) \\ C' = (C \setminus \{p\}) \cup \{p'\} \\ R' = \{\langle op_j, (C_j \setminus \{p\}) \cup \{p'\} \triangleleft p \in C_j \triangleright C_j \rangle\}_{1 \le j \le m} \\ S' = \langle n_S, I_S, O_S, C', R' \rangle \end{array}}{S' \sqsubseteq_A S}$$

Removing input channels. If the behaviour of a component p does not depend on the input from an input channel, then the channel can be removed. For a given system $S = \langle n_S, I_S, O_S, C, R \rangle$, let $p \in C$ be a component, then[7]

$$\frac{\begin{array}{c} p = \langle n_p, U_p, \overline{\alpha}_p : U_p \to (\mathsf{B}(U_p \times O_p) + \mathbf{1})^{I_p}, u_p \in U_p \rangle \\ i' \in \mathsf{in}.p \\ \forall u \in U_p \,.\, \alpha_p(u,i') = * \\ I'_p = I_p \setminus \{i'\} \\ p' = \langle n_p, U_p, \overline{\alpha}_{p'} : U_p \to \mathsf{B}(U_p \times O_p)^{I'_p}, u_p \in U_p \rangle \\ \forall u \in U_p, i \in I'_p \,.\, \alpha_{p'}(u,i) = \alpha_p(u,i) \\ C' = (C \setminus \{p\}) \cup \{p'\} \\ R' = \{\langle op_j, (C_j \setminus \{p\}) \cup \{p'\} \triangleleft p \in C_j \triangleright C_j \rangle\}_{1 \le j \le m} \\ S' = \langle n_S, I_S, O_S, C', R' \rangle \end{array}}{S' \sqsubseteq_A S}$$

Adding new components. We can simply add a new component $\mathsf{nil} = \ulcorner \mathsf{id}_\emptyset \urcorner$ to a system, which does not change the global system behaviour. For a given system $S = \langle n_S, I_S, O_S, C, R \rangle$, we have

$$\frac{\begin{array}{c} \forall p \in C \,.\, \mathsf{name}.p \neq \mathsf{name}.\mathsf{nil} \\ C' = C \cup \{\mathsf{nil}\} \\ S' = \langle n_S, I_S, O_S, C', R \rangle \end{array}}{S' \sqsubseteq_A S}$$

Removing old components. Components may be removed from a system if it does not have output that affects the system. For a given system $S = \langle n_S, I_S, O_S, C, R \rangle$ and a component $p \in C$, we have

$$\frac{\begin{array}{c} \mathsf{out}.p = \emptyset \\ C' = C \setminus \{p\} \\ R' = \{\langle op_j, (C_j \setminus \{p\}) \triangleleft p \in C_j \triangleright C_j \rangle\}_{1 \le j \le m} \\ S' = \langle n_S, I_S, O_S, C', R \rangle \end{array}}{S' \sqsubseteq_A S}$$

[7] The monad $\mathsf{B} + \mathbf{1}$ specifies the possibility of partial behaviour of components, where $*$ is the only element in the singleton set $\mathbf{1}$.

Decomposing components. Sometimes we may want to change the hierarchical structure of a system. For example, a lift system might consist of a lift controller, several doors and buttons. Especially, in later phases of a system development, we might consider the glass-box view of the system, and thus need to expand components into architectures. For a given system $S = \langle n_S, I_S, O_S, C, R\rangle$ and a component $p \in C$, which has the same behaviour with the architecture $T = \langle n_T, I_T, O_T, C_T, R_T\rangle$, i.e., p is behaviour equivalent to T, we have

$$\frac{\begin{array}{c} p \sim [\![T]\!] \\ \forall q \in C, q' \in C_T \;.\; \mathsf{name}.q \neq \mathsf{name}.q' \\ \mathsf{out}.C_T \cap \mathsf{out}.C = \mathsf{out}.p \\ C' = (C \setminus \{p\}) \cup C_T \\ R' = \{\langle op_j, (C_j \setminus \{p\}) \cup \{\xi^T_{root}\} \triangleleft p \in C_j \triangleright C_j\rangle\}_{1\leq j\leq m} \cup R_T \\ S' = \langle n_S, I_S, O_S, C', R'\rangle \end{array}}{S' \sqsubseteq_A S}$$

Folding components. If $T = \langle n_T, I_T, O_T, C_T, R_T\rangle$ is a subarchitecture of a system $S = \langle n_S, I_S, O_S, C, R\rangle$, then we can fold it into a new component $p = \langle n_p, U_p, \overline{\alpha}_p : U_p \to \mathsf{B}(U_p \times O_p)^{I_p}, u_p \in U_p\rangle$, which has the same behaviour with T.

$$\frac{\begin{array}{c} p \sim [\![T]\!] \\ C_T \subseteq C \\ \forall q \in C \setminus C_T \;.\; \mathsf{name}.q \neq n_p \\ C' = (C \setminus C_T) \cup \{p\} \\ R'' = (R \setminus (R_T \cup \{\langle op_j, C_j\rangle \mid \exists p' \in C_T \cap C_j\})) \\ R' = R'' \cup \{\langle op_j, (C_j \setminus C_T) \cup \{p\}\rangle \mid \exists p' \in C_T \cap C_j \;.\; \langle op_j, C_j\rangle \in R\} \\ S' = \langle n_S, I_S, O_S, C', R'\rangle \end{array}}{S' \sqsubseteq_A S}$$

5 Conclusions

This paper discusses refinement of software architectures in the context of a broader research agenda on coalgebraic semantics for componentware. From our experience to date, the appropriateness of the coalgebraic approach for component based systems is driven by the following two key ideas: first, the 'black-box' characterization of software components favors an *observational* semantics; secondly, the proposed constructions are *generic* in the sense that they do not depend on a particular notion of component behaviour. This led both to the adoption of coalgebra theory [26] to capture observational semantics and to the abstract characterization of possible behaviour models (*e.g.*, partiality or different degrees of non-determinism) by strong monads acting as parameters in the resulting calculus [4, 5]. Our work provides three basic refinement relations, which can be used for refinement of systems at different granularity.

A large body of work on software architectures using process algebraic ADLs can be found in the literature (see, *e.g.*, [10, 28]). These approaches are usually

biased towards specific behavioural models and therefore less generic than the one sketched in this paper. An approach closer to ours is that of [22], where a refinement mapping is defined to provide a syntactical translation between abstract and concrete architectures. However, such a mapping is required to be *faithful*, which means that both the positive and the implicit negative facts in the abstract architecture should be preserved in the concrete one. This makes both definition and proof of refinement difficult. Yet another interesting calculus was proposed in [24] to deal with refinement of information flow architectures. However, it only deals with the refinement of system's internal organization.

Our work is based on some preliminary results on behavioural refinement of generic state-based components documented in [19]. In this paper we proved a completeness result connecting simulation to behavioural refinement and provided further insight on refinement at both interface and architectural levels. Both of them can be reduced to the simple behavioural refinement relationship. A family of refinement rules was provided for local, stepwise modification of architectural designs. The genericity of the underlying coalgebraic model makes our approach not limited to any sort of architecture style. Whether it scales up to more sophisticated architectural models, namely the ones based on component coordination by anonymous communication and independent connectors (as in, *e.g.*, [2] or [17, 16]), is still an open research question.

References

1. Aynur Abdurazik. Suitability of the UML as an Architecture Description Language with Applications to Testing. Technical Report ISE-TR-00-01, Information and software engineering, George Mason University, 2000.
2. Farhad Arbab. Abstract Behavior Types: A Foundation Model for Components and Their Composition. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: First International Symposium, FMCO 2002, Leiden, The Netherlands, November 2002, Revised Lectures*, volume 2852 of *LNCS*, pages 33–70. Springer, 2003.
3. Farhad Arbab and Jan Rutten. A coinductive calculus of component connectors. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *Recent Trends in Algebraic Development Techniques: 16th International Workshop, WADT 2002, Frauenchiemsee, Germany, September 24-27, 2002, Revised Selected Papers*, volume 2755 of *LNCS*, pages 34–55. Springer-Verlag, 2003.
4. Luís Soares Barbosa. Towards a Calculus of State-based Software Components. *Journal of Universal Computer Science*, 9(8):891–909, August 2003.
5. Luís Soares Barbosa and José Nuno Fonseca de Oliveira. State-based components made generic. In H. Peter Gumm, editor, *Elect. Notes in Theor. Comp. Sci. (CMCS'03 - Workshop on Coalgebraic Methods in Computer Science)*, volume 82.1, Warsaw, April 2003.
6. Luís Soares Barbosa, Sun Meng, Bernhard K. Aichernig, and Nuno Rodrigues. On the semantics of componentware: a coalgebraic perspective. In Jifeng He and Zhiming Liu, editors, *Mathematical Frameworks for Component Software.- Models for Analysis and Synthesis*, chapter 2. World Scientific, 2004. To be published.

7. Alex Berson. *Client/Server Architecture*. McGraw-Hill, 1992.
8. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1999.
9. Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.
10. David Garlan. Higher-order connectors. Proceedings of Workshop on Compositional Software Architectures, January 1998.
11. C. A. R. Hoare, He Jifeng, and Jeff W. Sanders. Prespecification in data refinement. *Information Processing Letters*, 25:71–76, 1987.
12. Charles Antony Richard Hoare. Proof of correctness of data representations. *Acta Information*, 1:271–281, 1972.
13. John Hunt. *The Unified Process for Practitioners: Object Oriented Design, UML and Java*. Practitioner. Springer, 2001.
14. Paola Inverardi and Alexander L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Transactions on Software Engineering*, 21(4), 1995.
15. Bart Jacobs and Jesse Hughes. Simulations in coalgebra. In H. Peter Gumm, editor, *Elect. Notes in Theor. Comp. Sci. (CMCS'03 - Workshop on Coalgebraic Methods in Computer Science)*, volume 82, pages 245–263, Warsaw, April 2003.
16. M. A. Marco A. Barbosa and Luís Soares Barbosa. A Relational Model for Component Interconnection. *Journal of Universal Computer Science*, 10(7):808–823, July 2004.
17. M. A. Marco A. Barbosa and Luís Soares Barbosa. Specifying software connectors. In K. Araki and Z. Liu, editors, *1st International Colloquium on Theorectical Aspects of Computing (ICTAC'04)*, pages 53–68, Guiyang, China, September 2004. Springer Lect. Notes Comp. Sci. (3407).
18. Sun Meng and Luís Soares Barbosa. On Refinement of Generic Components. Technical Report 281, UNU/IIST, May 2003.
19. Sun Meng and Luís Soares Barbosa. On Refinement of Generic State-based Software Components. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Algebraic Methodology And Software Technology, 10th International Conference, AMAST'04, Proceedings*, volume 3116 of *LNCS*, pages 506–520. Springer, 2004.
20. Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.
21. Carroll Morgan. *Programming from Specifications, Second Edition*. Prentice Hall, 1994.
22. Mark Moriconi, Xiaolei Qian, and R. A. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, 1995.
23. OMG. *OMG Unified Modeling Language Specification, Version 1.3* , 2000.
24. Jan Philipps and Bernhard Rumpe. Refinement of information flow architectures. In M. Hinchey, editor, *Proceedings of ICFEM'97*. IEEE CS Press, 1997.
25. James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley Longman, 1999.
26. Jan Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249:3–80, 2000.
27. J.-G. Schneider and O. Nierstrasz. Components, scripts, glue. In L. Barroca, J. Hall, and P. Hall, editors, *Software Architectures - Advances and Applications*, pages 13–25. Springer-Verlag, 1999.
28. Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.