A Framework for Describing Software Architectures for Reuse*

Ezra Kaahwa Mugisa[1] and Tom S. E. Maibaum[2]
[1]Department of Mathematics and Computer Science; University of the West Indies (Mona); Kingston 7, Jamaica; phone/fax : +876 977 1810; e-mail: ekmugisa@uwimona.edu.jm:
[2]Department of Computing; Imperial College of Science, Technology and Medicine; 180 Queen's Gate, London SW7 2BZ, UK; phone : +44 171 594 8274; fax : +44 171 581 8024; e-mail : tsem@doc.ic.ac.uk

Key words: Systematic software reuse, software architectures, object calculus, category theory, component, interconnection, objects, patterns

Abstract: We present a formal description of software architectures for software reuse to support a view of systematic software reuse as the plugging of components into an architecture. The components are object descriptions in the object calculus. Interconnection between the components is defined via synchronisation morphisms within a framework based on category theory. Component composition is defined via the pushout construction, giving the architecture as a "calculated" component, from which the architecture's properties may be derived. We show that the architectures described are reusable in our Reuse Triplet that forms the motivation for our on-going work on systematic software reuse. This work provides further support for the suggestion that category theory provides the appropriate level of mathematical abstraction to describe software architectures.

* The work in this paper was supported by various grants from EPSRC and was partially sponsored by the project ARTS (formal Approach to Real-Time Systems development) being carried out by The Laboratory of Formal Methods, Department of Informatics, Pontificia Universidade Catolica do Rio de Janeiro, Brazil, and its research associates, under contract to Equitel SA Equipamentos e Sistemas de Telecomunicacoes. The first author, while on study leave at Imperial College, London, was supported by the University of the West Indies (Mona), Kingston, Jamaica, under the UWI/IDB staff development programme.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35563-4_35

1. INTRODUCTION

This work is motivated by a view of reuse-in-the-large that emphasises the reuse of software architectures. The importance of high-level abstraction to the success of reuse has been highlighted in the literature (Krueger, 1992; Biggerstaff and Richter, 1987). That the highest payoffs are to be expected from reusing high-level artefacts such as architectures has been well articulated by some authors e.g., (Krueger, 1992). The systematic reuse of analysis and design knowledge (encapsulated in software architectures) with potentially very high payoffs could help move reuse practice up towards the highest levels on a reuse maturity model.

In order to make reuse-in-the-large a reality, however, we need to have suitable ways of representing and reusing these large-grain software artefacts. Efforts have been made to find suitable ways of representing software architectures for reuse (Terry, et al., 1994; Gamma, et al., 1995; Tracz, 1995). However, one important problem still remains, namely how to find a good formal basis for component composition and interconnection in software architectures.

There are a number of formalisms in the literature for describing software architectures e.g., Darwin (Magee, et al., 1993) and Wright (Allen and Garlan, 1995). In the Darwin model a software architecture is described by a collection of Darwin components, each of which provides services to or requests services from its environment. Darwin components interact by having their service requests connected to appropriate service provisions. This is done by binding their corresponding ports. To instantiate a Darwin architecture, one simply instantiates its components.

In the Wright model an architecture is a collection of computational components together with a collection of connectors, that describe the interactions between the components. The Wright model differs from the Darwin model in that in the former, a connector is an explicit semantic entity. To instantiate a Wright architecture one instantiates the components and the connectors.

Fiadeiro and Maibaum (Fiadeiro and Maibaum, 1996) suggest that category theory provides the right level of mathematical abstraction to describe software architectures. Indeed they show that the category theory approach subsumes the Wright model of architectural description. It could be shown too that much of the Darwin model is similarly subsumed. We thus have a level of abstraction that appears to subsume other known architectural models, and appears to be suitable for performing formal analyses of software architectures in. Here we limit ourselves to issues of reusability.

We view systematic software reuse (SSR) as the process of identifying an appropriate reuse software architecture (RSA) and reuse software components (RSCs) and plugging the latter into the former. The RSA is a template with slots into which RSCs may be plugged. The template may be viewed as an abstraction of a family of systems with slots to be appropriately filled in for each specific system. We may express this view of reuse as the expression SSR = RSA ⊕ RSCs relating the Reuse Triplet (RSA, plugging, RSCs) or diagrammatically as in *Figure 1*. The plugging operator (⊕) takes a collection of reuse components and plugs them into the reuse architecture. The relationship between the RSA, RSCs and the target systems determines whether we are emphasising the reuse of RSAs or RSCs. In (Mugisa, 1997) we apply this view of reuse to well-known examples of reuse.

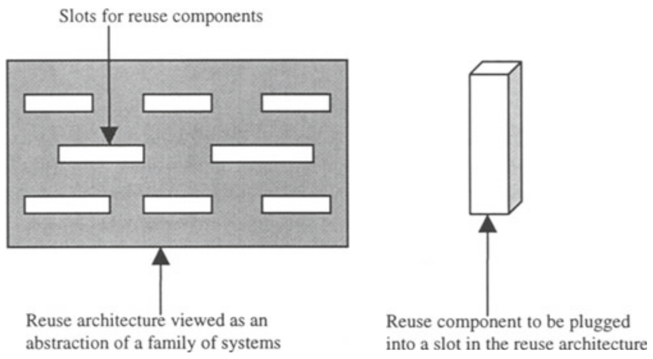

Slots for reuse components
Reuse architecture viewed as an abstraction of a family of systems
Reuse component to be plugged into a slot in the reuse architecture

Figure 1. A view of reuse: SSR = RSA ⊕ RSCs

In this paper we present a framework for a formal description of RSAs and RSCs. We have used the framework to describe the pattern-oriented software architectures of (Buschmann, et al., 1996). Here we will have space enough for presenting only one (simple) architecture. However, we have treated plugging and many of the more sophisticated examples discussed in the literature (Mugisa, 1998). We re-package the architectural patterns of (Buschmann, et al., 1996) at a level of abstraction that is consistent with our Reuse Triplet view. These architectural patterns interest us because the concept of a software pattern is deeply rooted in software reuse. After all a pattern recurs in several different applications from which it is abstracted. In describing architectural design patterns we are describing software architectures that have a high level of reusability.

One simple popular definition of a pattern is "a solution to a problem in a context." In (Gamma, et al., 1995) design patterns have been presented as "descriptions of communicating objects and classes that are customised to solve a general design problem in a particular context." Alexander, who initiated the pattern concept, has this to say about the patterns that he used to describe architectures of buildings (Alexander, et al., 1977) "Each pattern describes a problem that occurs over and over again in our environment, and then describes the core solution to that problem, in such a way that you can use this solution a million times over, without ever doing the same thing twice." Alexander's concept of a pattern is what we are interested in for systematic software reuse. The other two views tell us how to express patterns.

There are three parts to a pattern : context, problem, solution. We present the context as a domain theory in the object calculus. The problem is a specification in this domain theory; effectively as an extension of the domain theory. The solution takes components of the context domain theory and refines them to an appropriate level of detail. The underlying formal framework is provided by the object calculus : an appropriate temporal logic to express the properties of the basic components and category theory for interconnecting them and synchronising behaviours.

As an illustrative example, we apply our framework to the pipeline pattern from (Buschmann, et al., 1996). This and other architectures also appear in Mary Shaw's "popular architectural styles" (Shaw, 1995) and in Shaw and Garlan's "an emerging taxonomy of architectural styles" (Shaw and Garlan, 1996). It is difficult to present a more difficult example because of limitations of space, but the example used is examined in various versions, with the formalism helping to pinpoint the architectural differences and their resulting consequences/properties.

We would like to describe architectural styles in a way that enables us to reason about them so that we can determine interesting properties about them. Our motivation in doing this is well expressed by Mary Shaw (Shaw, 1995b) : "...although many design idioms are available, they are not clearly described or distinguished, and the consequences of choosing a style are not well understood."

Each software architecture will consist of roles for processing components and in some cases connecting components (connectors) to interconnect the processing components. We shall see how the type of processing and connecting components used affects the resulting architectural style. In all cases the interconnection between the components (either between processing components or between a processing component and a connector) will be described within a category of these components as suggested in (Fiadeiro and Maibaum, 1996). The instantiation of roles by reusable components is also expressed via morphisms in the underlying category (Fiadeiro and Lopez, 1997). A role is essentially a place holder for a processing component as seen from the connector. This is the view from a Wright connector (Allen and Garlan, 1995), for example.

We present each component as a theory description in the object calculus. Each of these theories is encapsulated as an object in the sense of (Fiadeiro and Maibaum, 1992). Each component then becomes an object description and at the same time a theory presentation following the spirit (if not the style) of (Fiadeiro and Maibaum, 1991, 1996).

2. INTERCONNECTING COMPONENTS

We present our architectures as interconnections of computational and connecting components. These components are viewed as theory descriptions represented as objects in Fiadeiro and Maibaum's object calculus (Fiadeiro and Maibaum, 1992). The interconnections are presented as diagrams in a category of these theory descriptions. Let us begin with some basic definitions and propositions.

DEFINITION 2.1 : *a-comp*

The components that are the basic building blocks of our architectures are object descriptions as defined in (Fiadeiro and Maibaum, 1992) and we call each of them an *a-comp* (*a*bstract *comp*onent). An a-comp is a (• , F) pair, where • is the component signature and F are the axioms of the component description.

A typical a-comp has the structure given in *Figure 2*, but there are variations. • = (<data_types>, <action_list>, <attribute_list>) and F = (<axioms>). An a-comp is treated as an object.

Component *a_comp_name*
  data types <*data_types*>
   actions <*action_list*>
   attributes <*attribute_list*>
   Axioms <*axioms*>
End

Figure 2. Structure of an a-comp specification

DEFINITION 2.2 : *sub-object*

Given two objects $\mathbf{obj}_1 = (\bullet_1, F_1)$ and $\mathbf{obj}_2 = (\bullet_2, F_2)$, $\mathbf{obj}_1$ is a sub-object of $\mathbf{obj}_2$ iff the behaviour described by object $\mathbf{obj}_2$ is an extension of the behaviour described by object $\mathbf{obj}_1$, in the sense that the principle of substitutability holds between them. The behaviour of an a-comp (which is an object) is constrained by the axioms F of the $(\bullet, F)$ pair. "The principle of substitutability says that if we have two classes, A and B, such that class B is a subclass of class A (perhaps several times removed), it should be possible to substitute instances of class B for instances of class A in *any situation* with *no observable effect*" (Bud, 1997). The term *subtype* is also used to describe this relationship. This sub-object relation is a form of inheritance for extension, possibly after renaming. It is the inverse of the sub-class relation. This definition should make subsequent discussion of synchronisation based on sharing a common sub-object more intuitive and less confusing. We have the following:

- the signature of $\mathbf{obj}_2$ $(\bullet_2)$ extends a signature isomorphic to the signature of $\mathbf{obj}_1$ $(\bullet_1)$
- F_2 is an extension of F_1, taking into account any renaming that may have been introduced
- $\mathbf{obj}_2$ is a sub-type of $\mathbf{obj}_1$, possibly after renaming

These concepts are expressed through the morphisms of the underlying category.

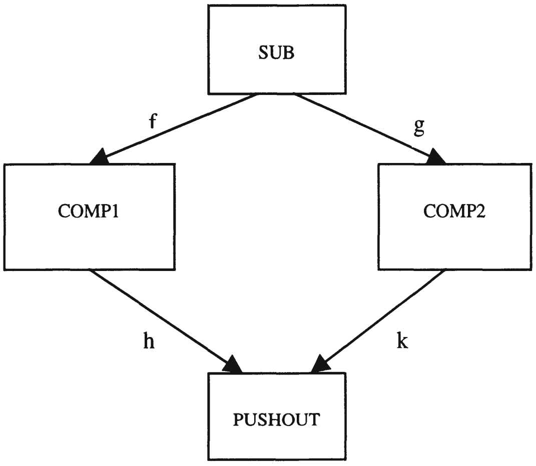
PROPOSITION 2.3 : *Category a-COMP*

The *a-comp*s and *sub-object* morphisms between them constitute a category *a-COMP*. A sub-object morphism transforms the source object into a sub-object of the target object.

- sub-object morphisms compose, as does inheritance

- there is an identity sub-object morphism - each *a-comp* is a sub-object of itself
- the sub-object relation is associative, as is inheritance for extension

The morphisms may use renaming and so the sub-objects may only be so identified after reverse renaming. We model component composition as a pushout construction within category *a-COMP*. In *Figure 3* component COMP3 is the result of synchronising components COMP1 and COMP2 on their common sub-object SUB, through sub-object morphisms f, g, h and k. SUB is shared by both COMP1 and COMP2, which coalesce around it to form COMP3.



$$\begin{tikzcd} & \text{SUB} \arrow[dl, "f"'] \arrow[dr, "g"] & \\ \text{COMP1} \arrow[dr, "h"'] & & \text{COMP2} \arrow[dl, "k"] \\ & \text{PUSHOUT} & \end{tikzcd}$$

Figure 3. Pushout construction in category a-COMP

DEFINITION 2.4 : *Interconnection*

The style of interconnection that we present here was motivated by the one used in (Fiadeiro and maibaum, 1992). The interconnection of a-comps is governed by the following :

- two components interconnect on a shared sub-object
- the shared sub-object has complementary (or dual) behaviour, e.g a plug and a socket, and it is this duality that makes the interconnection intuitive

This style of interconnection is used elsewhere, e.g., in binding requirements to provisions in the Darwin component abstraction (Magee, et al., 1993). We can view the *Provide* and *Request* ports in a Darwin component interconnection as manifestations of a shared sub-object - a provide/request port.

PROPOSITION 2.5 : *Communication via Ports*

All communication between a component and its environment will be channelled through its communication ports. This is a common model. We use two kinds of ports : ports for message passing (e.g., purely for data transfer) and (possibly implicit) ports for object invocation (see below). A port's *put* and *get* actions transfer data to and from the port's channel, respectively, as defined in Object *Port* below. The port may be viewed as a wrapper around the ubiquitous channel. Channel of T is a channel type capable of transferring data of type T.

```
Object Port
  data types Data, channel of Data
  actions get(Data), put(Data)
  attributes d : Data; ch : channel of Data
  axioms
    get(d) ⇒Xd = ch
    put(d) ⇒Xch = d
 End
```

This port abstraction enables synchronous communication as in CSP (Hoare, 1985) , Occam (INMOS, 1988) and Manna and Pnueli's ("no buffering") channels (Manna and Pnueli, 1992). Asynchronous communication is via a buffer between the communicating components.

DEFINITION 2.6 : *Object Invocation*

Object invocation here follows the CORBA request semantics (OMG, 1996) which states that "when a client issues a request, a method of the target object is called. The input parameters passed by the requester are passed to the method and the output parameters and return value (or exception and its parameters) are passed back to the requester." We use action *request(service-request)* for the action of the source (client) object. The argument (*service-request*) contains the requested service (or method) and parameters. In response, the target object will provide the requested service (if it can) and return results via its arguments. We use action *invoke* for the entire operation covering the request and the response.

Each invocation can in fact be adequately modelled by DMS synchronisation (definition 2.8), with service requests going from DMS-C1's output port to DMS-C2's input port and results going in the opposite direction (see the DMS synchronisation diagram in Figure 5). However, when there are several concurrent service requests, the model becomes messy and the *invoke* abstraction clears away the details. The result is MIS synchronisation as defined elsewhere (Mugisa, 1998).

2.1 We have three sub-objects

We use three types of sub-objects to interconnect the a-comps in our architectures. They correspond to the three ways in which we bind the components. These are Single Port Message-Passing Synchronisation (SMS), Double Port Message-Passing Synchronisation (DMS) and Multi-Port Invocation Synchronisation (MIS). *SMS-Sub, DMS-Sub* and *MIS-Sub* are such minimal sub-objects that can represent a component's ports.

Object *SMS_Sub*
  data types Port
  attributes p : Port
End

Object *DMS_Sub*
  data types Port
  attributes p_1, p_2 : Port
End

SMS-Sub has one port. When used as a synchronising sub-object it takes on both input and output roles in the interconnected components in order to effect message passing from one component to the other. See SMS synchronisation morphisms for details.

DMS-Sub is the 2-port (input/output) version of *SMS-Sub*. It encapsulates two *SMS-Sub* sub-objects.

Object *MIS_Sub*
  data types Service
  attributes $q_1, \ldots, q_n$: Service
End

Sub-object *MIS-Sub* contains services that are mapped to service-requests or to service-provisions. The requests are serviced by the provisions after synchronisation. This sub-object synchronises those components that are linked by object invocation.

2.2 Synchronisation morphisms

The pushout construction synchronises the components on their common sub-object around which they coalesce to form the pushout. There are several structures that the pushout of two a-comps in category a-COMP may have including the structure shown in the component <*pushout_name*> below. Here we discuss SMS and DMS synchronisation morphisms.

DEFINITION 2.7 : *SMS Synchronisation Morphisms*

These define the interconnection of two SMS a-comps by synchronising them on their common SMS sub-object as shown in *Figure 4*. The sub-object morphisms $f_1 : \{p_0 \mapsto out\}$ and $f_2 : \{p_0 \mapsto in\}$ identify *in* and *out* as synchronisation points for the two components, while morphisms g_1 and g_2 are synchronisation morphisms on the sub-objects identified by f_1 and f_2. The synchronisation may be expressed by identifying (or coalescing) the two sub-objects as follows: $f_1 : \{out \mapsto p'\}$ and $f_2 : \{in \mapsto p'\}$. Here is the structure of an SMS a-comp (*SMS_COMP*).

Component *SMS_COMP*
  data types Data, Port
  actions
  attributes *in, out* : Port; *d* : Data
  Axiom
    Relevant behavioural axioms
End

Component *pushout_name*
  Inherit *a_comp1, a_comp2*
  Synchronisation Axioms
    Set of Synchronisation Axioms
End

Figure 4 is the interconnection diagram for SMS a-comps *SMS-C1* and *SMS-C2* yielding pushout *SMS-C3*. (In *SMS-C1*, the *get* action on port *out* is suppressed, i.e., axiom ¬(*SMS.C1.out.get*) holds. Similarly, axiom ¬ (*SMS.C2.in.put*) holds for *SMS-C2* to make *in* an input port.) So we actually use a *specialisation* of SMS_Sub in SMS-C1 (and SMS-C2); this is an example of a different form of reuse through inheritance, well known in object-oriented programming and design. This disabling property is maintained by the morphism g_1 (and g_2) as a property of the resulting system. The two components will synchronise on their respective ports, i.e., *SMS-C1.out.put* ≡ *SMS-C2.in.get*. This means that *SMS-C1*'s output action and *SMS-C2*'s input action become synchronised, thus effecting data flow between the two components. This is what interconnecting these two components is supposed to achieve. Synchronisation is expressed, in pushout component SMS_C3, by action *sync* on the relevant ports and by appropriately unifying actions on these ports.

We have introduced the colon notation as an alternative to qualification by the dot notation. We write *A : Exp* to mean that symbols in expression *Exp* are qualified by object *A*. Therefore we may write *SMS_C1 : out.put(d)* as shorthand for *SMS_C1.out.put(SMS_C1.d)*.

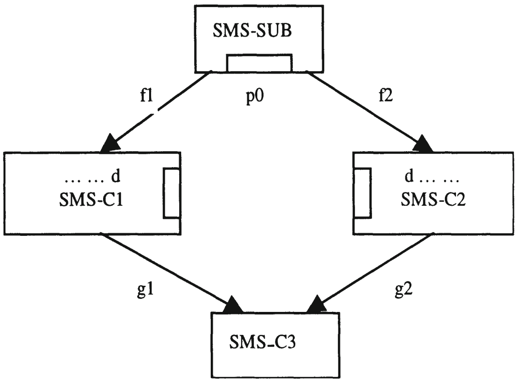


Figure 4. SMS synchronisation diagram

DEFINITION 2.8 : *DMS Synchronisation Morphisms*

These define the interconnection of two DMS a-comps by synchronising them on their common DMS sub-object as shown in *Figure 5*. A DMS a-comp is the double port version of an SMS a-comp. Sub-object morphisms $f_1 : \{p_1 \mapsto out_1 ; p_2 \mapsto in_1\}$ and $f_2 : \{p_1 \mapsto in_2 ; p_2 \mapsto out_2\}$ identify (out_1, in_2) and (in_1, out_2) as pairs of synchronisation points for the two components, while morphisms g_1 and g_2 are once again synchronisation morphisms in the sense

Component *SMS_C3*
  Inherit *SMS_C1, SMS_C2*
  Synchronisation Axiom
    sync(SMS_C1.*out*, SMS_C2.*in*)
    SMS_C1 : *out.put*(*d*) $\Leftrightarrow$ SMS_C2 : *in.get*(*d*)
End

described in definition 2.7.

In the diagram of *Figure 5*, a-comps *DMS-C1* and *DMS-C2* are interconnected to yield pushout *DMS-C3*. Action suppression applies as for SMS ports.

The synchronisation diagrams here (in *Figures 4* and *5*) are interpreted as follows:

- morphisms f_1 and f_2 are sub-object morphisms
- morphisms g_1 and g_2 are sub-object and synchronisation morphisms, coalescing the components on the sub-objects identified by f_1 and f_2.
- the pushout (X-C3) (where X is SMS or DMS) inherits both X-C1 and X-C2 and then coalesces them around their common sub-object. It adds synchronisation axioms that translate the synchronisation morphisms (g_1 and g_2) into "equivalence" axioms relating the synchronised attributes and actions.

In both cases the pushout is calculated as an a-comp that extends the inherited theories as discussed above.

Component *DMS_C3*
  Inherit *DMS_C1, DMS_C2*
  Synchronisation Axiom
    sync(DMS_C1.out_1, DMS_C2.in_2)
    sync(DMS_C1.in_1, DMS_C2.out_2)
    DMS_C1 : $out_1.put(d_1)$ $\Leftrightarrow$ DMS_C2 : $in_2.get(d_1)$
    DMS_C1 : $in_1.get(d_2)$ $\Leftrightarrow$ DMS_C2 : $out_2.put(d_2)$
End

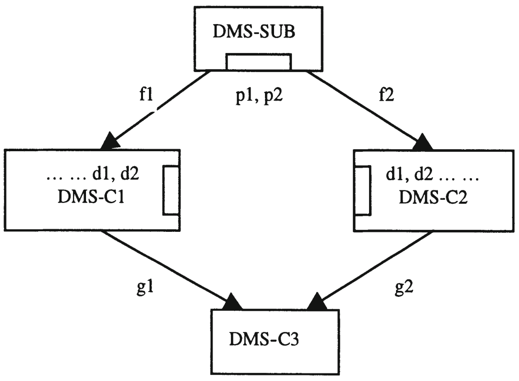


Figure 5. DMS synchronisation diagram

2.3 Asynchronous interconnection of components

In the SMS and DMS Synchronisation diagrams presented above the interconnection between X-C1 and X-C2 is synchronous, requiring the two components to rendezvous in order to communicate. In order to decouple them we interconnect them asynchronously through a buffer, for example as Manna and Pnueli have done in (Manna and Pnueli, 1992). We may use the same synchronisation diagrams to depict asynchronous binding if either X-C1 or X-C2 is a buffered connector. This is how we get the asynchronous connection of *Filter1* to *Filter2* via *Pipe1* in *Figure 6.*

For asynchronous DMS inter-connection we have the equivalent of two pipes going in opposite directions. The details are in (Mugisa, 1998).

2.4 A comparison between synchronous and asynchronous inter-connection of components

It has been stated by C.A.R. Hoare (Hoare, 1978) and others that synchronous communication is the more basic form of communication on top of which asynchronous communication may be implemented as buffered synchronous communication. The trade-off between synchronous and asynchronous inter-connection of components is in decoupling the connected components which must nevertheless separately rendezvous with their buffered connector. We have to depend on the properties of the connector to guarantee that we get the desired asynchronous behaviour from the same components that give us the desired synchronous behaviour.

Manna and Pnueli (Manna and Pnueli, 1992) point out that synchronous communication offers some advantages over the asynchronous version because the execution of a synchronous communication immediately provides the sender with an acknowledgement that the communication has taken place, whereas in the asynchronous case such an acknowledgement has to be explicitly "programmed". The liveness axioms in our buffered connectors give us the same guarantees as the synchronous case except for the delay. On the other hand the asynchronous connection with unbounded buffering gives the decoupled components freedom to exercise independent behaviour without giving up the general properties of the synchronous case except for the introduction of the delay as mentioned above.

3. THE PIPELINE ARCHITECTURE

The "ball of mud" class of architectures is one of those covered in (Buschmann et al., 1996). The ball-of-mud is to be transformed into an organised structure (or a system) by decomposing it into interacting and co-operating sub-systems. The mode of interaction is strongly linked to the chosen architecture. In our setting, a system is defined by an architecture and a set of sub-systems that are instances of more abstract sub-tasks. The ball-of-mud general context is presented below as a domain theory (object *B_O_M_Context*) that states that a system is constructed by plugging a set of sub-tasks into an architecture. The plug action replaces an abstract architectural sub-task by a concrete system sub-task.

```
Object B_O_M_Context
  data types Sub_Task, System, Architecture
  actions plug(Architecture, set of Sub_Task
  attributes arch : Architecture
  Axioms
    $\forall sys : System . \exists subs : set\ of\ Sub\_Task . sys = plug(arch, subs)$
End
```

In this paper we examine only one ball-of-mud architecture known as the pipeline (or pipe-and-filter) architecture. Shaw and Garlan in (Shaw and Garlan, 1996) have this to say about pipes and filters :

"In a pipe-and-filter style each component has a set of inputs and a set of outputs. A component reads streams of data on its inputs and produces streams of data on its outputs. This is usually accomplished by applying a local transformation to the input streams and computing incrementally, so that output begins before input is consumed. Hence components are termed *filters*. The connectors of this style serve as conduits for the streams, transmitting outputs of one filter to inputs of another. Hence the connectors are termed *pipes*."

Buschmann et al. in (Buschman et al., 1996) distinguish between active and passive filters and present four pipeline scenarios depending on whether the filters are passive-push (triggered by an active data source), passive-pull (triggered by an active data sink), passive/active-pull/push (triggered by an active filter pulling and pushing) or the more typical all active-pull-compute-push. The passive filters are triggered into a push/pull by direct calls or by data from neighbouring components. This removes the need for pipes and makes the resulting pipelines less interesting for reuse. We shall stick to the more reuse-friendly UNIX-like pipelines of active filters connected by pipes.

In the pipeline architecture the sub-tasks (filters) are arranged sequentially; the output of one sub-task is the input of the next one in the sequence. A pipe component asynchronously connects neighbouring filter components. This system may be specified by the diagram of *Figure 6*. Specifications for all the components of this architecture are presented in the next few sections. The pipeline context given by object *Pipeline_Context* below, simply states that all sub-tasks are filters and that all connectors are pipes.

Object *Pipeline_Context*
  Inherit *B_O_M_Context*
  data types Filter, Pipe
  Axioms
    Sub_Task $\mapsto$ *Filter*
    Architecture.Connector_type = Pipe
End

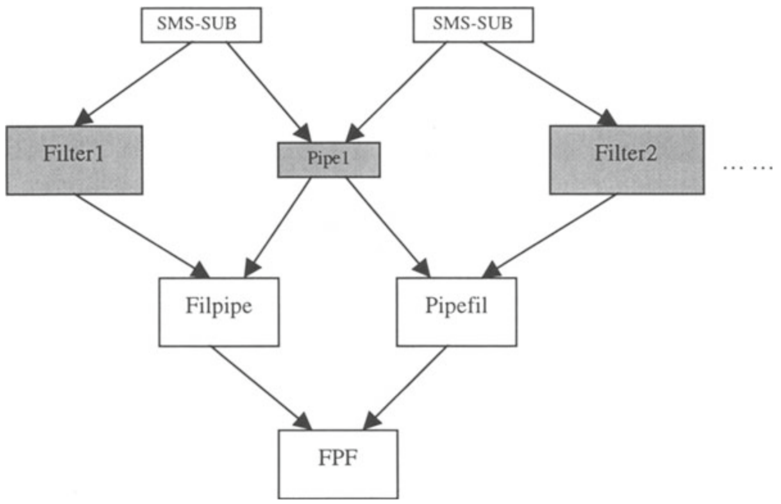


Figure 6. Specification of a system as a pipeline of filters and pipes (asynchronous)

3.1 The components

Component *Filter* below encapsulates our filter a-comp. The liveness axiom guarantees the desired *get-process-put* sequence of actions. Component *Pipe*, the connector, contains a buffer defined as a queue with actions *getq* and *putq*. The safety axioms respectively initialise the buffer, guarantee the absence of an unsolicited response from the buffer and protect buffer update. The second has been included only for emphasis. In our setting it is redundant because it is a consequence of the locality principle (Fiadeiro and Maibaum, 1991, 1992), that states that only the actions declared for an object can change the values of its attributes. For *Pipe*, this may be stated as $(getq \vee putq) \vee ((Xq = q) \wedge (Xin = in) \wedge (Xout = out) \wedge (Xd = d))$. The liveness axiom promises a guaranteed response from the non-empty buffer.

```
Component Filter
  data types Data, Port
  actions process(Data)
  attributes in, out : Port; d : Data
  Axioms
    $process(d) \Rightarrow \mathrm{X}d = processed(d)$
  liveness
    $in.get(d) \Rightarrow \mathbf{F}(process(d) \wedge \mathbf{XF}out.put(processed(d)))$
End
```

```
Component Pipe
  data types Buffer, Data, Port
  actions getq, putq
  attributes in, out : Port; q : Buffer; d : Data
  Axioms
    $getq \Rightarrow in.get(d) \wedge Xq = q \;@\; \mathrm{X}d$
    $putq \Rightarrow q \neq [\,] \wedge q = \mathrm{X}d :: \mathrm{X}q \wedge$
    $\mathrm{X}d = hd(q) \wedge \mathbf{XF}out.put(\mathrm{X}d)\}\}$
  safety
    $\mathbf{beg} \Rightarrow q = [\,]$
    $\neg getq \wedge \neg putq \Rightarrow \mathrm{Xq} = q$
    $\neg(getq \wedge putq)$
  liveness
    $q \neq [\,] \Rightarrow \mathbf{F}(out.put(first(q)))$
End
```

Instead of a buffered pipe we could have a single item mailbox thus allowing the consumer filter to lag behind its producer filter by at most one data item. This is discussed in section 3.2.2.

Yet another alternative is to have a synchronous pipeline with no pipe connecting the filters. The filters would then be synchronised directly, thus having to rendezvous in order to communicate. Section 3.2.3 has the details.

3.2 Interconnection diagrams

There are three versions of the interconnection diagram corresponding to the three ways of connecting the filters discussed above. We discuss in full the asynchronous version with a buffered pipe and then show how the other two differ from it.

3.2.1 Asynchronous interconnection diagram

To interconnect a *Filter* a-comp with a *Pipe* we use SMS synchronisation of definition 2.7. *Figure 6* is the categorical diagram that shows how two *Filter* components are connected by a *Pipe* in this way.

Components *Filpipe* and *Pipefil* are the local pushouts of the left-hand side and the right-hand side of the interconnection, respectively. They are instantiations of component SMS-C3 specified earlier.

Component *Filpipe*
  data types Filter, Pipe
  attributes *Filter1* : Filter; *Pipe1* : Pipe
  Synchronisation Axioms
    sync(Filter1.out, Pipe1.in)
    Filter1 : out.put(d) $\Leftrightarrow$ *Pipe1 : getq*
End

Component FPF is the pushout of the diagram in which *Pipe1* is the common sub-object of *Filpipe* and *Pipefil.* This component is a pushout of two other pushouts and not of simple components. This difference is reflected in the structure of its specification as a component that coalesces two structured components (pushouts) around a connector component as sub-object. The synchronisation axiom reflects this. FPF is also the colimit of the larger (Filter1, Pipe1, Filter2) diagram. All FPF components are similarly connected as suggested in *Figure 6* to define a final colimit (not shown) for the entire diagram.

```
Component FPF
  Instances Filpipe, Pipefil
  Synchronisation Axioms
    Filpipe.Pipe1 ≡ Pipefil.Pipe1
End
```

3.2.2 Asynchronous interconnection via a single item mailbox

We take a single item mailbox to be equivalent to a buffer of size 1. We define component *Mailbox_Pipe* similar to component *Pipe* with *buffer* replaced by *mbox*, the single item mailbox type. This connector enables the source filter to lag behind the target filter by only one item, i.e., no new item may be delivered until the previous item has been collected by the target filter.

```
Component Mailbox_Pipe
  data types mbox, Port
  actions getm, putm
  attributes in, out : Port; m : mbox
  Axioms
    getm ⇒ m = [] ∧ in.get(m)
    putm ⇒ m ≠ []∧ out.put(m) ∧ Xm = []
  safety
    beg ⇒ m = []
    ¬getm ∧ ¬putm ⇒ Xm = m
  liveness
    m ≠ []⇒ F(out.put(m))
End
```

3.2.3 Synchronous interconnection

A synchronous interconnection of two filter a-comps is a direct synchronisation of the a-comps. For a transfer of data to take place the two a-comps must rendezvous directly. This would be an instance of SMS synchronisation and is described by *Figure 7*. This is really like composing two functions directly. The tight coupling between the two filter components is evident from the second binding axiom of component FF.

Component *FF*
  data types Filter
  attributes *Filter1, Filter2* : Filter
  Synchronisation Axioms
    sync(Filter1.out, Filter2.in)
    Filter1 : out.put(d) ⇔ *Filter2 : in.get(d)*
End

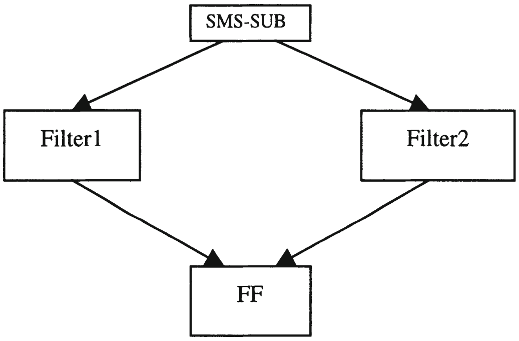


Figure 7. Synchronous pipeline connection diagram

3.3 The pipeline architecture has the pipeline property

In a pipeline connection, if a data item appears at the input port of the first (left) processing component it will eventually appear at the output port of the second (right) processing component. We would like to prove that equation (1) below holds in component FPF. We use the shorthand colon notation introduced earlier as an aid to readability. We get the definitions of *get* and *put* from proposition 2.5. From the definition of *get* we get (1) and from *Filter*'s liveness axiom we get (2). From (1), (2), the synchronisation axioms of *Filpipe* and *getq* for *Pipe*, the value at port *Filter1.in* through attribute *d* and port *Filter1.out* has been added to buffer *Pipe.q*. From *Pipe*'s liveness axiom we shall get (3). From *Pipefil*'s synchronisation axiom and *Filter2*'s liveness axiom we shall eventually get (4) which is what we are required to prove.

$$\text{Filter1} : \text{in.get}(d) \Rightarrow \text{Filter2} : \text{Fout.put}(\text{processed}(\text{Filter1}:\text{processed}(d))) \quad (1)$$

$$\text{Filter1} : \mathbf{X}d = \text{in.ch} \quad (2)$$

$$\text{Filter1} : \mathbf{F}(\text{process}(d) \wedge \mathbf{XF}(\text{out.put}(\text{processed}(d)))) \quad (3)$$

$$\text{Pipe} : \text{Fout.put}(\text{first}(q)) \quad (4)$$

$$\text{Filter2} : \text{Fout.put}(\text{processed}(\text{Filter1}:\text{processed}(d))) \quad (5)$$

Furthermore, it can be proved (using queue operations *getq, putq*) that because we have used a queue to buffer incoming data in the connector, incoming data will be in the order in which it is output by the previous filter in the pipeline.

Since component FPF can be reduced to the structure of *Filter*, if we combine the sequence of *Filter1.process*, the buffer operations and *Filter2.process* into one process, thus also hiding the buffer attribute (and of course its actions) then we get a *Filter*. So we can extend the pipeline to any length we want.

3.4 The architecture

Figure 6 represents the pipeline as an object (the colimit) and as an asynchronous connection of filters. Let us call it component *Pipeline_Arch*. Two other similar figures for the mailbox and synchronous connections can be deduced from *Figure 6*, giving three versions of the architecture. Let us call their representative objects *Pipeline_Arch_Buffered, Pipeline_Arch_Mailbox, Pipeline_Arch_Synchronous*. We therefore have the following expression for the pipeline architecture:

Pipeline_Arch ::= Pipeline_Arch_Buffered | Pipeline_Arch_Mailbox | Pipeline_Arch_Synchronous

3.5 Is the pipeline architecture reusable?

If the pipeline architecture, *Pipeline-Arch*, is an RSA in the Reuse Triplet then it can be reused by plugging in appropriate RSCs. In *Figure 8* RSCs *$RealFilter_1$* and *$RealFilter_2$* are plugged into the two *Filter* slots of a pipeline RSA to produce resultant component *RealFPF*. The plugging morphisms are k_1 and k_2. A similar diagram for the synchronous version, can be deduced easily.

The plugging operator must satisfy the requirement that important properties of the RSA are preserved after each slot instantiation. We shall not go further into this topic here, but it has been covered at length in (Mugisa, 1998) under plugging. *Figure 8* appears to contain objects from two categories - the category *a-COMP* and the category of instantiations of a-comps, ie, programs (Fiadeiro and maibaum, 1995, 1997). The existence of a functor between the two categories suggests a way forward.

An alternative way of showing that the pipeline architecture is indeed reusable is to focus on the connector as the central piece in the architecture and to view the components it connects together as its parameters or roles (as the roles of (Allen and Garlan, 1995; Fiadeiro et al., 1997)). We may then show that the connector is reusable by constructing a diagram in which a role (*Filter*) is mapped to its instantiation (*RealFilter*) and completing the diagram with its pushout, component *RealFPF*. See *Figure 8*.

In the plugging diagram of Figure 8, morphisms k_1 and k_2 are the instantiation morphisms of connector *PIPE*'s roles. Morphisms k_{11}, h_1 and k_{12} on the left and k_{21}, h_1' and k_{22} on the right serve to plug the slots in the RSA using the given instantiations. Of course, there is also a role in the connector, ie the buffer. An instantiation of this role by an appropriate buffer implementation would result in an enlarged system (described by a new colimit extending realFPF). Alternatively, one could regard the buffer as an already implemented part of the architecture (hence, describing a less reusable architecture) and represented in the architecture description by the image of the program under the functor that maps programs to their corresponding (minimal, canonic) specifications (Fiadeiro and Maibaum, 1995, 1997).

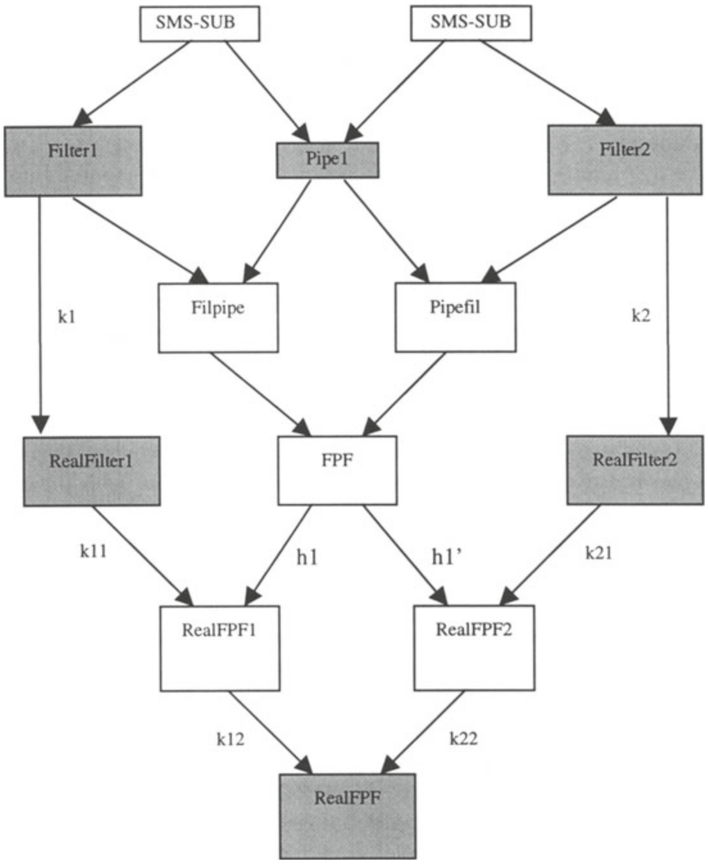


Figure 8. Instantiating a pipeline connection

3.6 Linking context, problem, and solution

We would like to fit *Pipeline_Arch* within the *Pipeline_Context* domain theory and be reassured that all the pieces fit together consistently.

mapping data types : *B_O_M_Context.Sub_Task* $\mapsto$ *Pipeline_Context.Filter*

defining data types : *Pipeline_Context.Filter; ::= Component Filter*

problem : *System ::= plug(*$subtas_1$ .. $subtask_n$*, Architecture)* solution : *B_O_M_Context.Architecture ::= Object Pipeline_Arch*

The result of all this is given below in Object Pipeline_Context_Solution and in Figure 9.

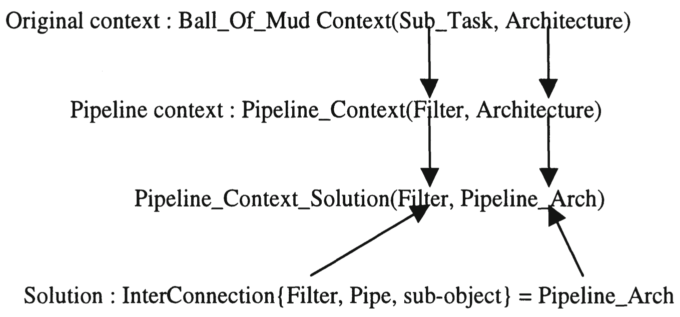

Original context : Ball_Of_Mud Context(Sub_Task, Architecture)
Pipeline context : Pipeline_Context(Filter, Architecture)
Pipeline_Context_Solution(Filter, Pipeline_Arch)
Solution : InterConnection{Filter, Pipe, sub-object} = Pipeline_Arch

Figure 9. Linking context, problem and solution

Object *Pipeline_Context_Solution*
  Inherit *Pipeline_Context*
  data types Filter, Pipeline_Arch
  Axioms
    Pipeline_Context.Filter $\mapsto$ Filter
    Architecture $\mapsto$ Pipeline_Arch
End

4. RELATED WORK

Shaw and Garlan in (Shaw and Garlan, 1995) discuss the inadequacy of present formalisms to deal adequately with important issues of software architecture. They give examples of Wright and Darwin. We mentioned these formalisms in our introduction. Whereas Wright (with its CSP base) permits static checks such as deadlock freedom as Allen and Garlan did in (Allen and Garlan, 1995), it does not appear to be suitable for issues of dynamic architecture, component composition and interconnection. On the other hand much of the strength of Darwin (with its π-calculus base) is in being able to deal with issues of dynamic configuration of architectures as for example Magee and Kramer have done in (Magee and Kramer, 1996).

The work of Abowd, Allen and Garlan (Abowd et al., 1993) also attempts to formalise and reason about software architecture. We think that the chosen formalism, Z (Spivey, 1992), is not appropriate. The notion of structure in Z (which revolves around the schema) is rather weak and is not suitable for studying what is, after all, a problem of structure. The notion of schema is used more to talk about the textual structure of a specification rather than inherent structure and interconnection as in 'configuring a system out of components'. Also Z does not allow us to talk about behaviours (of components), only about input/output specification of individual operations. To allow us to talk about behaviour, we would need a version of Z that incorporated a temporal logic.

In an effort to find a formalism that adequately deals with interconnection and composition of components for software architecture, Fiadeiro and Maibaum in (Fiadeiro and Maibaum, 1995, 1997) suggested category theory and showed how it subsumed Wright. The work reported in this paper builds on that of Fiadeiro and Maibaum. Our categorical framework also uses as its foundation the work of Joseph Goguen on interacting objects (Goguen, 1991, 1992), especially the principle that "interconnecting systems corresponds to taking colimits in the category of systems, where sharing is indicated by inclusion maps from shared parts into the systems that share them" (Goguen, 1992). Corresponding to Goguen's systems and inclusion maps are what we have called components and sub-object morphisms between them. The general nature of Goguen's categorical framework (as expressed in (Goguen, 1992)) has made our successful application of it to issues of software architecture less surprising than it might have been.

The Kestrel Institute's SpecWare (Srinivas, 1995) is a tool that supports the modular construction of formal specifications and the stepwise and componentwise refinement of such specifications into executable code. Srinivas and McDonald in (Srinivas and McDonald, 1996) report that one of the formal foundations of SpecWare is category theory. They report that the language of category theory has produced for SpecWare a highly parameterised, robust and extensible architecture that can scale to system-level software construction. The colimit operation is their main tool for composing structures, in particular by "gluing" together parts that have overlaps. We use the same operation here to define interconnection of components of an architecture (which are themselves formal specifications) via our sub-object morphism. The operation is the same - "glue" together components along the sub-component that they share.

Rapide's architecture view of "wired interfaces" (Luckham et al., 1995) can be given a categorical semantics through our framework. Since Rapide derives many of its concepts from VHDL (Perry, 1998), we can easily accommodate VHDL's configurations. Rapide's interface and VHDL's entity are abstracted to our component (or slot), simple or structured. Rapide's wired interfaces and VHDL's configurations of connected entities are models of our interconnected components. Mapping entities to their implementing architectures in VHDL's configurations and tying modules to interfaces in Rapide are what we call plugging in our framework. The details of how our framework relates to Rapide and VHDL (two prototyping languages for software and hardware, respectively) are covered elsewhere (Mugisa, 1998).

The work of Moriconi and others (Moriconi et al., 1994, 1995) on architecture refinement is more closely related to plugging in our framework and that is presented elsewhere (Mugisa, 1998). However, architecture composition, which they touch on briefly in (Moriconi and Qian, 1994) is appropriately handled by our framework since an architecture may act as a component of a larger architecture.

5. CONCLUSION

We have presented a framework for describing software architectures for reuse (or RSAs). We present the components of the architectures (or a-comps) as object descriptions in the object calculus. We describe the interconnections between the a-comps using sub-object morphisms between them in a category *a-COMP* of component specifications. An RSA is then derived as the pushout of a categorical diagram that shows how the a-comps are interconnected. This gives us a formal technique for composing (or "calculating") an architecture from its constituent a-comps. We can then derive architectural properties from the resultant a-comps. We have presented an example whereby we used this framework to describe the pipeline architecture and were able to prove one desirable property of this architecture - we called it the pipeline property. We have used the framework to describe other RSAs as well but there is no space in this paper to report on those. In other related work we examine the plugging operator of the Reuse Triplet.

The mathematical underpinnings for this technique were laid out by Fiadeiro and Maibaum in (Fiadeiro and Maibaum, 1995, 1997). In this paper we have applied the mathematics to an engineering problem, namely composing an architecture from (its) components (and with the same structure) as long as we can identify common constituent parts on which to synchronise the components. We can then analyse the resultant architecture using the same tools used on the components it is derived from.

REFERENCES

Abowd, Gregory; Allen, Robert and Garlan David (1993), Using Style to Understand Descriptions of Software Architecture, *ACM SIGSOFT '93 : Foundations of Software Engineering, Software Engineering Notes,* 18(5). ACM Press

Alexander, Christopher; Ishikawa, Sarah; Silverstein Murray; Jacobson Max ; Fiksdahl-King Ingrid and Angel, Shlomo (1977), *A Pattern Language*, Oxford University Press.

Allen Robert and Garlan David (1995), Formalizing Architectural Connection, *First International Workshop on Architectures forSoftware Systems.*

Biggerstaff, Ted and Charles Richter, Charles (March 1987), Reusability Framework, Assessment, and Directions, *IEEE Software.*

Bud, Timothy (1997), *An Introduction to Object-Oriented Programming, Second Edition*, Addison Wesley

Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad, Peter and Stal, Michael (1996), *Pattern-Oriented Software Architecture A system of Patterns*, John Wiley & Sons.

Fiadeiro, J and Maibaum, T (1991), Describing, Structuring and Implementing Objects, *LNCS: Foundations of Object-Oriented Languages*, Volume 489, (de Bakker, J. W.; W. P. de Roever, W. P. and Rozenberg, G, editors), Springer-Verlag,

Fiadeiro, J and Maibaum, T (1992), Temporal Theories as Modularisation Units for Concurrent System Specification, *Formal Aspects of Computing*, 4(3).

Fiadeiro, J and Maibaum, T (1997), Categorical Semantics of Parallel Program Design, *Science of Computer Programming*, North Holland.

Fiadeiro, J. L and Lopez, A.(1997), Semantics of Architectural Connectors, *TAPSOFT '97: Theory and Practice of Software Development, LNCS 1214*, (Bidoit, Michel and Dauchet, Max, editors), pages 505 - 519, Springer-Verlag.

Fiadeiro, J. L.; Lopez, A. and Maibaum, T. (1997), Synthesising Interconnections, *IFIP TC2 Working Conference on Algorithmic Languages and Calculi*, (Smith, D. and Finance, J.-P., editors), Chapman and Hall.

Fiadeiro, J and Maibaum, T (1995), Interconnecting Formalisms: Supporting Modularity, Reuse and Incrementality, *Proceedings of the 3rd Symposium on Foundations of Software Engineering*, (Kaiser, G.E., editor), ACM Press.

Fiadeiro, J and Maibaum, T (1996), A Mathematical Toolbox for the Software Architect, *Proceedings of the 8th International Workshop on Software Specification and Design*, IEEE Press.

Gamma,Erich;Helm,Richard;Johnson,Ralph and Vlissides, John (1995),*Design Patterns*,Addison-Wesley.

Goguen, Joseph (1991), A Categorical Manifesto, *Mathematical Structures in Computer Science*, 1(1).

Goguen, J. A. (1992), Sheaf semantics for concurrent interacting objects, *Mathematical Structures in Computer Science*, 2(2), Pages 159 - 191.

Hoare, C.A.R.(1985), *Communicating Sequential Processes*,Prentice Hall.

Hoare, C.A.R.(1978), Communicating Sequential Processes, *Communications of the ACM*,21(8),

INMOS-Ltd (1988), *Occam 2 Reference Manual,* Prentice Hall,

Krueger, Charles (1992), Software Reuse, *ACM Computing Surveys*, 24(2).

Luckhaın,David C.; Kenney, John J.; Augustin,Larry M.; Vera, James; Doug, Bryan and Mann, Walter (1995), Specification and Analysis of System Architecture Using Rapide,*IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4),

Luckham,David C. and Vera (1995), James, An Event-Based Architecture Definition Language, *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(9).

Magee,Jeff; Dulay, Naranker and Kramer, Jeff (1993),Structuring Parallel and Distributed programs,*IEE Software Engineering Journal*, 8(2).

Magee, Jeff Kramer, Jeff and Sloman, Morris (1989), Constructing Distributed Systems in Conic, *IEEE Transactions on Software Engineering*, 15(6),

Manna, Zohar and Pnueli, Amir (1992*), Temporal Logic of Reactive and Concurrent Systems : Specification*, Springer-Verlag.

Moriconi, Mark and Qian, Xiaolei (1994), Correctness and Composition of Software Architectures, *ACM SIGSOFT '94 : Symposium on Foundations of Software Engineering*, Software Engineering Notes (Wile, David (ed.)).

Moriconi, Mark; Qian, Xiaolei and Riemenschneider, R. A.(1995), Correct Architecture Refinement, *IEEE Transactions on Software Engineering*, 21(4),

Mugisa, Ezra Kaahwa (1997) , A Reuse Triplet for Systematic Software Reuse, *Software Engineering Notes*, 22(2),

Mugisa, Ezra Kaahwa (1998), *An Approach to Systematic Software Reuse Based on Plugging Components into an Architecture*, PhD thesis, University of London, in preparation.

Object Management Group (1996), *The Common Object Request Broker:Architecture and Specification, Revision 2.0.*

Perry, Douglas L.(1998), *VHDL , Third Edition*, McGraw-Hill.

Shaw, Mary (1995), Architectural Issues in Software Reuse: It's Not Just the Functionality, It's the Packaging, *Proceedings of the Symposium on Software Reusability (SSR'95)*, (Samadzadeh, Mansur and Zand, Mansour (eds.)), Software Engineering Notes.

Shaw, Mary (1995),Comparing Architectural Design Styles,*IEEE Software*, 12(6).

Shaw, Mary and Garlan, David (1995), Formulations and Formalisms in Software Architecture, Computer Science Today: Recent Trends and Developments. *Lecture Notes in Computer Science 1000* (Jan van Leeuwen (ed.)), Springer-Verlag.

Shaw, Mary and Garlan, David (1996), *Software Architecture: Perspectives on an emerging discipline*, Prentice Hall.

Spivey, J. M.(1992), *The Z Notation : A Reference Manual*, Second Edition, Prentice Hall.

Srinivas, Y. V. and Jullig, Richard (1995), *Specware(TM): Formal Support for Composing Software*, Kestrel Institute Technical Report KES.U.94.5

Srinivas, Yellamraju V. and McDonald, James L. (1996), *The Architecture of Specware, a Formal Software Development System*, Kestrel Institute Technical Report KES.U.96.7

Terry, Allan; Hayes-Roth, Frederick and Erman, Lee (1994), Overview of Teknowledge's Domain-Specific Software Architecture Program, *Software Engineering Notes*, 19(4)

Tracz, Will (1995), Domain Specific Software Architecture, *Software Engineering Notes*, 20(3)