Pushout: A mathematical model of architectural merger

Andrew Solomon

Faculty of Information Technology, University of Technology, Sydney
Email: andrews@it.uts.edu.au
August 4, 2006

Abstract. Although there are many formal representations of architecture, actually determining what an architecture should be when systems are merged is largely based on context and human intuition. The goal of this paper is to find a *mathematical* model which supports this context and determines the architecture when the systems have been merged. A category of architectural models is presented, and the pushout in this category provides the *unique minimal* merger of two architectures by way of an abstraction of the desired intersection. We conclude by identifying deeper aspects of architectural type which should be incorporated into this theory, and how the whole model might be automated.

1 Introduction

An Architecture Description Language (ADL) provides means for a formal model of systems' components and their connections. A model clarifies the purpose of the components and their interactions and, for the most part engineers represent these architectures using graphs [18], such as UML architectural diagrams [5].

It often happens (for example, when companies merge) that their computer systems with overlapping functionality also need to be merged with minimal duplication of functionality in the resulting system. Because the architecture of two systems merged is not a simple cut-and-paste, one hopes for formal and well-defined ways to merge system designs without errors.

1.1 Related Work

Modelling systems has been addressed at many levels from syntax, through logic based theories to abstract graph representations. At each level, maintaining the known properties of systems being merged is a core issue.

At the syntactic level, the process of merging two modifications of the same code has been dealt with extensively using the well established computation of text differences. Although the process of merging them has been semi-automated in projects such as the ArchStudio version control system [3], determining the merger is far from obvious. Niu et al. [14] have modelled the code as a graph labelled by Fuzzy logic values (which are essentially a partially ordered set) and merging the models of two programs has been implemented using the categorical pushout.

At the architectural level, merger has been attempted for systems modelled with a logic based ADL. Moriconi and Qian [13] merge two architectures by a union of their theories and provide a method for determining whether a composition of two systems is *faithful*, which is to say there is no collapse in the separation of components apart from those specified. Using logic to define the smallest architecture containing a set of properties, Caporuscio et al. [2] give a method to test whether the theories are contradictory. While this is not specifically about system merger, it could clearly be a bottom-up approach. With a logic based model of program specifications, Goguen and Burstall [10] provide a very similar categorical approach as presented below, using the colimit (a generalization of pushout) to merge several specifications. In a slight deviation from the logic approach, algebras have also been used to model software specifications [11] with merger being defined to be the pushout when certain conditions are met.

In spite of its rigour, logic does not fit comfortably in an engineer's intuitive graphical approach. The purpose of this paper is to formulate a graph based ADL as a mathematical model which accommodates some of the *context* of the systems being merged [6]. Le Metayer [12] uses a graph grammar to describe the process of adding and removing components of a system, while Baresi et al. [1] use graph homomorphisms to model architectural abstraction. Modelling architecture by graphs labelled by a poset of component types, Denford et al. [4] give an approach to refine an abstract description into a model closer to the implementation level. As with Fahmy and Holt [7,8] one of the main topics is an abstraction with only the parts of the architecture relevant to the activity at hand – in our case, the systems being merged.

1.2 Contents of this Paper

In Section 2 we illustrate an example of systems to be merged, together with the obvious intuitive solutions. Section 3 formalizes the notion of connection and component *types* and gives a mathematical representation for the relationships of their attributes. An architectural type (or as we refer to it, *archetype*) is presented in Section 4 as a graph of component and connection types. Together with the components and connections themselves, an architecture is then defined in Section 5 as a graph projected upon an archetype by graph morphism.

In Section 6 the examples will demonstrate that on this categorical basis, the merging of two architectures using pushout seems close to human intuition. Furthermore, the fact that it is a pushout means that it is the unique and smallest architecture which contains both of its sub-architectures.

In the conclusion we summarize how the aims of this paper have been achieved and identify possible approaches to issues yet to be addressed such as automation.

2 Three Examples of Merging

Suppose two companies have merged and they need their two IT systems (Figure 1) to become a single system.

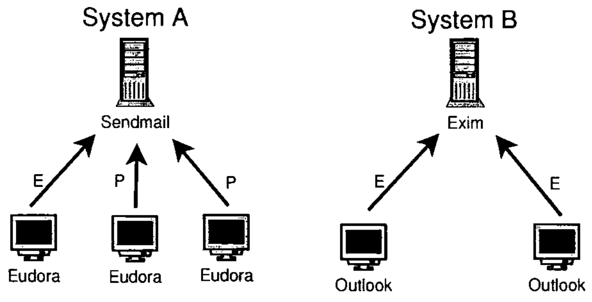

Fig. 1. Email services of two companies, with SMTP connections over Ethernet (E) or PPP (P)

The differences between the two systems account for different activities, for example Sendmail has already been setup for dealing with email over dialup connections (PPP), while Exim has not, and to change (according to management) would be an unnecessary expense. In this sense, Sendmail has more attributes of value to the new organization.

Regarding the client side, many claim [16] that Eudora (relative to Outlook) has serious inefficiency with large folders but is better for control filters.

Generally, there are three ways the new IT team might solve this problem which are depicted in Figure 2.

The first solution is easy but may make any retrenchment of employees infeasible. The second solution is possible because the *Sendmail* server has all the necessary functionality. Having both *Eudora* and *Outlook* may keep the users happy as they both have distinct but useful features, but it would be an expense to the company to have two different client packages to buy and maintain. Therefore, the third solution would be to seek out an entirely different email client which satisfies all the staff requirements.

For the remainder of this paper it is shown that these three approaches can be formally modelled so that known system requirements are satisfied and which ensures that the erroneous models (such as replacing Sendmail with Exim) cannot occur.

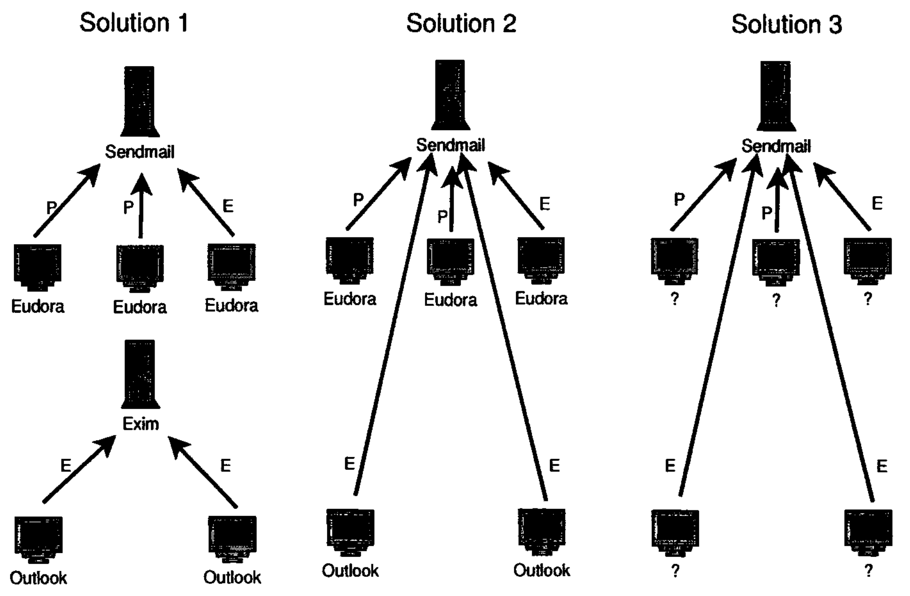

Fig. 2. Three possible email services of the new company where E is an Ethernet based SMTP connection and P is a PPP based SMTP connection.

3 Posets of Types

In the examples there are two sets of types: *component types* such as Exim and Sendmail, and *connection types* - SMTP over PPP (P) versus SMTP over the Ethernet (E).

There is no exact definition of these types other than the code which implements them. But rather than giving up, we propose formalizing relations of these types based on the attributes which are important to the stake-holders of the given situation.

A *partially ordered set* or *poset* is a set X together with a binary relation (which we write as $\leq$) which is reflexive (for all $x \in X$, $x \leq x$), antisymmetric (for all $x, y \in X$, $x \leq y$ and $y \leq x$ implies that $x = y$) and transitive (for all $x, y, z \in X$, $x \leq y$ and $y \leq z$ implies that $x \leq z$). Most importantly, in a *partially* ordered set it may be the case that for some $x, y \in X$, neither $x \leq y$ nor $y \leq x$.

We can form a poset Π of the component types and a poset Λ of the connection types by the definition that $x \geq y$ if and only if x has all the attributes of y in the given situation. We depict the posets related to our example in Figure 3 using a graph with an arrow $x \dashrightarrow y$ meaning $x \geq y$. The basis of this work is that if $x \geq y$ in the poset of component types, a component of type y can be replaced with a component of type x. Similarly for connection types. Furthermore, writing *Outlook* $\vee$ *Eudora* we mean "some minimal application which covers both of their attributes".

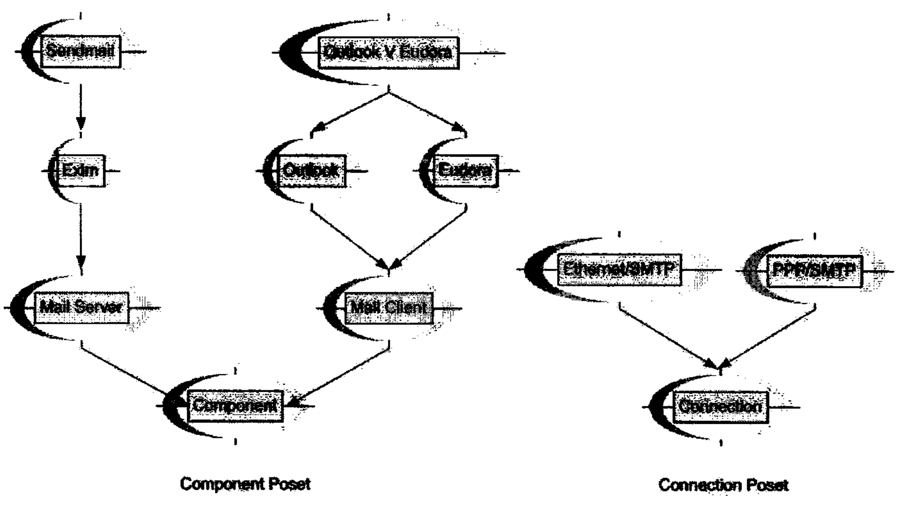

Fig. 3. Posets of component and connection types

4 Archetypes and Architectures

In this section we extend the notion of type from components and connections to architectures. A architecture's type (in short, an *archetype*) is simply a graph labelled by elements of the posets of component and connection types. An archetype is not in itself an architecture, but merely the description of the component and connection types which exist in an architecture. For example, the archetypes of the architectures depicted in Figure 2 are given in Figure 4.

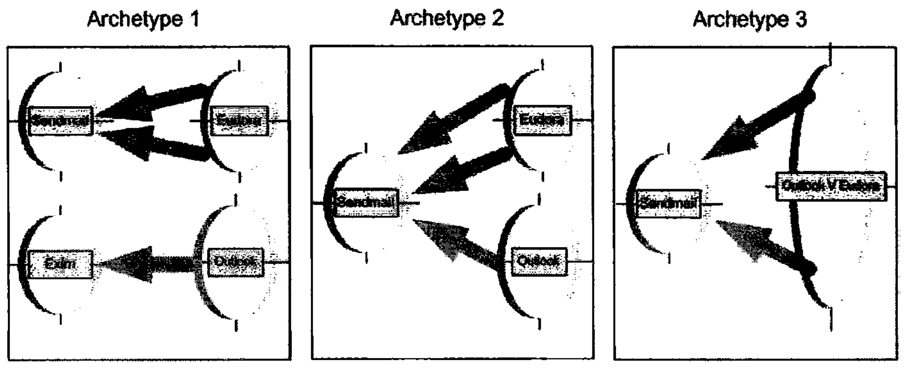

Fig. 4. Archetypes of Solutions 1, 2 and 3 in Figure 2.

Now, we show an *architecture* to be a graph morphism from an unlabelled graph (the actual components and their connections) to the graph of its archetype, which defines the types. The following section gives a concise formalization of these notions, and their consequences through category theory. Thenceforth, we are able to show the main result - a formal model of merging architectures.

5 A Category of Architectures

As introduced in Section 3, Π is a poset of component types and Λ is a poset of connection types. An *archetype* (also known as a ***poset labelled graph***) G is a tuple $(V_G, E_G, s_G, t_G, \pi_G, \lambda_G)$ where V_G and E_G are sets of *components* and *connections* respectively; $s_G, t_G : E_G \to V_G$ define the *source* and *target* of a connection; and $\pi_G : V_G \to \Pi$ and $\lambda_G : E_G \to \Lambda$ are the types of the components and connections.

A *morphism* $\phi : G \to H$ of archetypes is a pair $(\phi_V : V_G \to V_H, \phi_E : E_G \to E_H)$ such that for all $e \in E_G$: $s_H(\phi_E(e)) = \phi_V(s_G(e))$ and $t_H(\phi_E(e)) = \phi_V(t_G(e))$ meaning ϕ preserves the structure of the graphs; and such that for all $x \in V_G, e \in E_G$, ϕ has *lax* preservation of the types, that is:

$$\pi_H(\phi_V(x)) \geq \pi_G(x) \text{ and } \lambda_H(\phi_E(e)) \geq \lambda_G(e) \qquad (1)$$

– a component (connection) of one type can only be mapped to a component (connection) of greater or equal type as per the posets of types. In case the mappings in Equation 1 are all equalities, we say that ϕ is *strict*.

It is easy to see that the composition of morphisms, as pairs of functions, is another morphism. Associativity and identity are inherited from the category of sets and functions. Therefore, with posets Π and Λ fixed as above, archetypes (poset labelled graphs) and their morphisms form a category which we denote by $\mathsf{Graph}_{\Pi,\Lambda}$.

5.1 Formalization of Architecture

Let $U : \mathsf{Graph}_{\Pi,\Lambda} \to \mathsf{Graph}$ be the functor which forgets the labelling of an archetype, and consider the comma category Graph/U. An object of Graph/U is a pair $(T, X : G \to UT)$ (often simply written as X) we call an *architecture*. The architecture X consists of a typed graph T, called the *archetype*, and the graph G, called the *component* graph, equipped with the graph morphism X from G to the underlying ordinary graph of T. An arrow (*architectural morphism*) is a pair (f, t) such that the following diagram commutes

$$\begin{tikzcd} G \arrow[r, "f"] \arrow[d, "X"'] & H \arrow[d, "Y"] \\ UT \arrow[r, "U(t)"] & UT' \end{tikzcd}$$

Definition 1. *Given a category* $\mathsf{Graph}_{\Pi,\Lambda}$ *of archetypes and the functor* $U : \mathsf{Graph}_{\Pi,\Lambda} \to \mathsf{Graph}$*, then the comma category* Graph/U *is called a* category of architectures.

Informally identifying T and UT, for any component v of G define the component $X(v)$ of T to be the *archetype* of v while $\pi(X(v)) \in \Pi$ is the *type* of v. Define archetype and type similarly for connections.

Let $c : \mathsf{Graph}/U \to \mathsf{Graph}$ map an architecture to its graph of *components and connections*, and let $a : \mathsf{Graph}/U \to \mathsf{Graph}_{\Pi,\Lambda}$ map an architecture to its *archetype*. It can be shown that there is a natural transformation $l : c \to Ua$.

Figure 5 illustrates the way that objects of Graph/U represent architectures. The top part of the diagram is the object in Graph and the bottom part is its archetype in $\mathsf{Graph}_{\Pi,\Lambda}$.

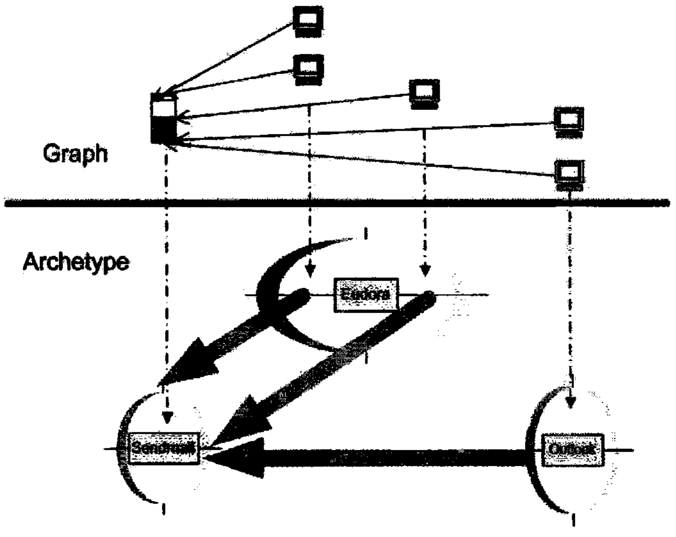

Fig. 5. An architecture – an object of Graph/U

6 Merging Architectures by Pushout

As a quick reminder before launching into the pushout of Graph/U consider the first example of pushout in most textbooks – the union of sets.

For example, let $X = \{1,2,3\}$ and $Y = \{2,4,6\}$ then the union of these two sets, denoted $X \cup Y$ is equal to $\{1,2,3,4,6\}$. This is, the unique smallest set containing the elements of both X and Y. One can express this category theoretically by saying, let $f : Z \to X$ and $g : Z \to Y$ be injective functions, which define Z as the *intersection* of X and Y. Then the *pushout* is an object P

together with a pair of arrows $i_1 : X \to P$ and $i_2 : Y \to P$ such that the inner square of Figure 6 *commutes* (that is, $i_1 f = i_2 g$), and furthermore, that given any other diagram (j_1, j_2, Q) there is a unique arrow from P to Q making the whole diagram commute. (In the example of sets, this unique arrow is simply indicating that any set which contains both X and Y contains $\{1,2,3,4,6\}$, but could be larger.)

Therefore, the pushout is a formal construct that ensures the containment is complete, unique (up to isomorphism) and minimal. In the remainder of this section we present the mathematical details of the pushout for architectures.

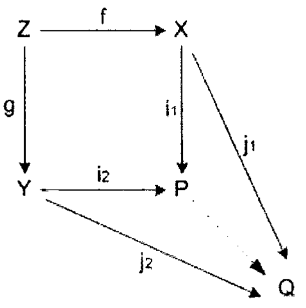
$$\begin{tikzcd}
Z \arrow[r, "f"] \arrow[d, "g"'] & X \arrow[d, "i_1"] \arrow[ddr, "j_1"] & \\
Y \arrow[r, "i_2"] \arrow[drr, "j_2"'] & P \arrow[dr, dotted] & \\
 & & Q
\end{tikzcd}$$

Fig. 6. Pushout diagram

It is a straightforward consequence of [15, Lemma 3.9] that

Proposition 1. $\mathsf{Graph}_{\Pi,\Lambda}$ *has pushouts along strict monomorphisms.*

and more generally

Theorem 1. *In* Graph/U *there are pushouts along arrows* η *if its archetype part* η_a *is a strict monomorphism.*

However in trying to model merger, it is not helpful to demand that η_a be strict, so instead we assert that Π and Λ have least upper bounds. It can be shown that

Theorem 2. *If* Π *and* Λ *have least upper bounds, then* $\mathsf{Graph}_{\Pi,\Lambda}$ *has pushouts (and, in fact, all its colimits) and the architecture category* Graph/U *has all pushouts (and all its colimits).*

Note that although real-world component and connection types will not generally have least upper bounds, the join operation (for example *Outlook* $\vee$ *Eudora* of Figure 3) produces any necessary types which can then be analysed to determine how they will be implemented in practice. Therefore we can now formalize what it means to merge architectures:

Definition 2. *The* merger *of architectures* X *and* Y *over the archetype* Z *with monomorphisms* $X \xleftarrow{i} Z \xrightarrow{j} Y$ *is the pushout of* X *and* Y *over* Z.

In the next section we use this framework to identify two subsystems of the merging systems which are similar enough that a single subsystem can replace them both in the architecture of the merge.

6.1 Three Examples of Merging (again)

It should now be clear that the examples in Section 2 are each a pushout in the category of architectures. Solution 1 (see Figure 2) comes about when the intersection architecture is empty. Solution 2 is the pushout when the intersection architecture is a single Email Server. The archetype pushout ensures that as this maps to both Sendmail server and an Exim server, the result is a Sendmail server. The most difficult example (Archetype 3 in Figure 4) is illustrated in Figure 7 where a least upper bound appears as the join of *Outlook* and *Eudora*, alerting one to the decision which needs to be made on the new type of these components.

7 Conclusion

The goal of this paper was to find a mathematical model of architecture which is intuitive, yet rigorous when determining architectural merger. By "intuitive" it is meant to be close to the way in which engineers think about architectures and this has been done by using graph-like representations.

The rigour required for merging two systems is to maintain the structure of both systems while avoiding any unnecessary duplication. This was achieved by formalizing the contextual and intuitive definition of types and implementing the merger as a categorical pushout. At the core of this process is identifying the intersection of the systems as different parts which, by abstraction, are the same.

Although this is a well controlled model of types, there are many sets of attributes which apply to an archetype - and not merely the sum of the attributes of its parts. For instance, there are several properties determining *architectural style* [9] such as ensuring that a particular type of connection has only one server to many clients. It may therefore be helpful to move from types to a single (infinite) poset of archetypes (together with some constraint on the comma category) creating a more detailed definition of architecture.

An interesting project would be to modify a graph transformation based program such as PROGRES [8] or AGG [17] to incorporate architectures as defined in this paper and automate the pushout once the intersection is chosen by the user.

References

1. L. Baresi, R. Heckel, S. Thöne and D. Varrò, *Style-Based Refinement of Dynamic Software Architectures*, In Proc. WICSA 2004: 4th Working International IEEE/IFIP Conference on Software Architecture Page(s) 155-164, Publisher: IEEE Computer Society.

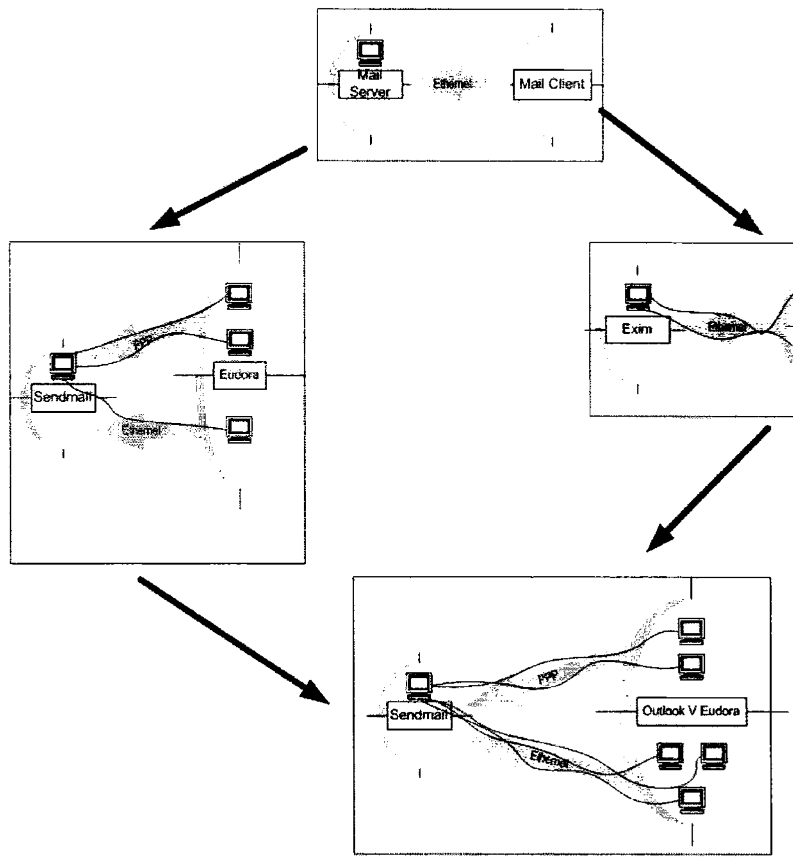

Fig. 7. 'Top-down' view of a pushout diagram in Graph/U, where an architecture is encapsulated in a box. The component graph (the terminal icons) sit above the archetypes (circular objects) which define their types. The pushout results in Solution 3 in Figure 2. Notice that if this diagram were from $\mathsf{Graph}_{\Pi,A}$ it would not be a pushout and therefore not have the useful properties of being minimal and unique.

2. Mauro Caporuscio, Paola Inverardi and Patrizio Pelliccione, *Formal Analysis of Architectural Patterns*, EWSA 2004, LNCS 3047, pp. 10–24, 2004.
3. Eric M. Dashofy, André van der Hoek, Richard N. Taylor, *Towards architecture-based self-healing systems* Proceedings of the first workshop on Self-healing systems Charleston, South Carolina 21 – 26 (2002) ISBN:1-58113-609-9
4. Mark Denford, Andrew Solomon, John Leaney and Tim O'Neill, *Architectural Abstraction as Transformation of Poset Labelled Graphs*, Journal of Universal Computer Science, Special Issue on Formal Specification of Computer Based Systems, Journal of Universal Computer Science, vol. 10, no. 10 (2004), 1408-1428.
5. Martin Fowler and Kendall Scott, "UML Distilled", Second Edition, Addison Wesley Longman, Inc. 2000.

6. R. Kazman, G. Abowd, L. Bass, and P. Clements, *Scenario-Based Analysis of Software Architecture*, IEEE Software, pp.47-55, November 1996.
7. Hoda Fahmy and Richard C. Holt, *Software Architecture Transformations*, ICSM 2000: International Conference on Software Maintenance, San Jose, CA, Oct 2000.
8. Hoda Fahmy and Richard C. Holt, *Using Graph Rewriting to Specify Software Architectural Transformations*, p.187, 15th IEEE International Conference on Automated Software Engineering (ASE'00), 2000.
9. David Garlan, Robert T. Monroe, and David Wile, *Acme: Architectural Description of Component-Based Systems* Foundations of Component-Based Systems, Gary T. Leavens and Murali Sitaraman (eds), Cambridge University Press, 2000, pp. 47–68.
10. Joseph A. Goguen and Rod M. Burstall, *Institutions: abstract model theory for specification and programming*, Journal of the ACM, vol. 39, No. 1, January 1992, pp. 95 – 146.
11. A. E. Haxthausen and F. Nickl, *Pushouts of Order-Sorted Algebraic Specifications*, In Algebraic Methodology and Software Technology (AMAST'96), number 1101 in Lecture Notes in Computer Science, pages 132–148. Springer, 1996.
12. Daniel Le Métayer, *Describing Software Architecture Styles Using Graph Grammars*, IEEE Transactions on Software Engineering, vol. 24, No. 7, July 1998.
13. Mark Moriconi and Xiaolei Qian, *Correctness and Composition of Software Architectures*, Proceedings, ACM SIGSOFT'94, Symposium on Foundations of Software Engineering, New Orleans, Louisiana, December, 1994, pp. 164-174.
14. Nan Niu, Steve Easterbrook, Mehrdad Sabetzadeh, *A Category-theoretic Approach to Syntactic Software Merging*, 21st IEEE International Conference on Software Maintenance, pp. 197-206.
15. F. Parisi-Presicce, H. Ehrig and U. Montanari, *Graph rewriting with unification and composition*, in 'Graph grammars and their application to computer science', Lecture Notes in Computer Science (1986), 496–514.
16. Google Groups Discussion Forum, *comp.mail.eudora.ms-windows.*
17. The Attributed Graph Grammar System, `http://tfs.cs.tu-berlin.de/agg/index.html`, Last Modified: 30 January 2006.
18. Carnegie Mellon University, Software Engineering Institute, `http://www.sei.cmu.edu/architecture/adl.html`, Last Modified: 31 August 2005.