Journal of Universal Computer Science, vol. 10, no. 10 (2004), 1408-1428
submitted: 9/7/04, accepted: 17/10/04, appeared: 28/10/04 © J.UCS

Architectural Abstraction as Transformation of Poset Labelled Graphs

Mark Denford
(University of Technology, Sydney, Australia
Mark.Denford@uts.edu.au)

Andrew Solomon
(University of Technology, Sydney, Australia
andrews@it.uts.edu.au)

John Leaney
(University of Technology, Sydney, Australia
John.Leaney@uts.edu.au)

Tim O'Neill
(University of Technology, Sydney, Australia
Tim.ONeill@uts.edu.au)

Abstract: The design of large, complex computer based systems, based on their architecture, will benefit from a formal system that is intuitive, scalable and accessible to practitioners. The work herein is based in graphs which are an efficient and intuitive way of encoding structure, the essence of architecture. A model of system architectures and architectural abstraction is proposed, using poset labelled graphs and their transformations. The poset labelled graph formalism closely models several important aspects of architectures, namely topology, type and levels of abstraction. The technical merits of the formalism are discussed in terms of the ability to express and use domain knowledge to ensure sensible refinements. An abstraction / refinement calculus is introduced and illustrated with a detailed usage scenario. The paper concludes with an evaluation of the formalism in terms of its rigour, expressiveness, simplicity and practicality.

Keywords: Architecture, Refinement, Abstraction, Graphs
Categories: D.2.1, D.2.2, F.3.1, F.4.2

1 Introduction

1.1 Issues with Computer Based Systems

Computer Based Systems (CBSs) are typically large, complex, real-time, highly functional, resource sharing, distributed systems [Lavi 1991, Lawson, et al. 1999, Rowe 1999]. Due to the inherent complexity of CBSs they regularly incur long and difficult design and implementation processes, and the resulting systems often fall short of the required attributes, especially non-functional attributes (performance, reliability, openness, security etc). Frequently, these problems result from poor design [Neumann 2004].

Developing representations of the architecture of CBSs promises to alleviate some of these problems [Horowitz 1991, Rechtin 1991]. For instance, many non-functional attributes can be calculated once the architecture of the system is known [O'Neill, et al. 2000, Payne 1999, Rowe and Leaney 1997], and with a suitable representation this can be done *prior* to building the system, when errors are significantly easier and less costly to fix [Sommerville 2001].

Based on consulting work undertaken by some of the authors with large Australian organisations [Avolution 2004], a typical architecture would contain thousands of components of type Database, Application, Server, and Network and tens of thousands of connections of type Information Flow, Database Request, Network Connection.

Case studies do exist to demonstrate the successful application of traditional formal software engineering methods to large scale engineering problems [van Lamsweerde 2000, Wordsworth 1991]. However, the organisations performing these case studies are always highly skilled engineering and computer science organisations. The authors' consulting work has been with organisations such as financial services (banks and insurance companies) and fast moving consumer goods (FMCG). Their experience suggests that these formal methods [Abrial, et al. 1979, Guttag, et al. 1993, Hoare 1985, Jones 1980] are too complex to be applied usefully in these organisations. The motivation is thus to develop an appropriate, lightweight, formal system that can be used by the typical engineer.

1.2 Motivations

The motivations for this work are several. A realistic stance recognises that it is not possible to fully automate the role of the designer and eliminate the human element during design. There is a great need for a formal, yet practical system that can be used by the human designer. One must recognise the difficulties that many practitioners face in using traditional formal methods based in predicate logic and other similarly complex mathematics [Abrial, et al. 1979, Guttag, et al. 1993, Hoare 1985, Jones 1980] and this provides part of the motivation for the work presented herein.

One must also recognise the context in which the designer works. No system is ever built from nothing: in practice the designers will have suggestions for the system at every level of abstraction [Alexander 1964, McMenamin and Palmer 1984, Ward and Mellor 1985]. Consequently, the design of a "new" system will often start with some understanding of the design of other systems. As a consequence, this work is also motivated by the desire to develop a formalism that allows the designer to begin their design activities with an architecture at any level of abstraction. The formalism should also be usable in a variety of different design situations. For example, the designer may take a concrete architecture and wish to abstract it to an appropriate level of abstraction where certain analyses and modifications can be made, and then refine again to a new concrete architecture [Denford, et al. 2003]. The formalism should support this.

A designer often has expert knowledge of a certain domain, and the work is motivated by the desire to allow the designer to encode this knowledge within the constructs of the formalism.

Finally, while the formalism should be practical in nature, and flexible in its use, it should also be rigorous with respect to the correctness of the architectures being

refined. From one level of abstraction to another certain properties of the architectures should be maintained and the designer should be able to rely on this being the case. With a formalism that largely concentrates on the structure of the architecture, the properties that are maintained are statements about the connectedness of the abstract and refined architectures.

Therefore, the formalism should be practical, flexible, expressive but still rigorous.

1.3 Informal Notion of Architecture

The IEEE 1471 standard presents an informal definition of architecture, which states that *architecture* is: "the fundamental organisation of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution" [IEEE 2000]. IEEE 1471 also states that "components and connectors may be typed" [IEEE 2000].

1.4 What is to be Formalised

This work aims to formalise two related concepts, both exhibited in the informal IEEE definition above. The first is to formalise the notion of *architecture*. The second is to formalise one of the primary principles in guiding the design of architectures: *architectural refinement*. Refinement in this context is the act of taking a high-level abstract architecture and successively refining it into a concrete and more detailed architecture. The concept of designing at different levels of abstraction and refining is understood in many areas of computer science and engineering: for instance program proving and software development [Abrial, et al. 1979, Ward and Mellor 1985], software architecture [Bass, et al. 2003], and systems architecture [Rechtin 1991].

It should be noted that architecture is often regarded as more than just structure, and can also include behavioural and narrative information [Bass, et al. 2003, Shaw and Garlan 1996]. This work deals mainly with the structural aspects of architecture and architectural refinement; however, it does model type within the architecture, which provides some narrative information.

1.5 Related Work

Various models and formalisms for architecture have been presented, though perhaps without explicit reference to the informal IEEE definition. We give a brief survey this work, grouped according to its principal concern.

1.5.1 Behaviour

Modelling the behaviour of a system allows for certain analyses such as the detection of deadlock and livelock. Allen presents a formal model for architecture based on the WRIGHT architecture description language (ADL) [Allen 1997]. It is a "formal description of the abstract behaviour of architectural components and connectors" with behavioural information modelled using communicating sequential processes (CSP) [Hoare 1985]. Refinement of the architectures is achieved primarily through refinement of the CSP specifications using mathematical proof obligations. Allen presents a model of a Client in a Client-Server system however the specification is

complex and unintuitive. This tends to discourage the use of this sort of specification for reasons of practicality. In fact, Allen goes on to highlight that this "simple" specification is not actually complete, and the complete specification is more complex still.

1.5.2 Dynamic Structure

Many architectures change at runtime: components and connections may be dynamically created, bound, and destroyed. This is referred to as the dynamic structure of the system. Bolusset and Oquendo [Bolusset and Oquendo 2002] model architectures using an ADL based on the pi-calculus by Milner [Milner 1993] which models the dynamic nature of the architecture under runtime reconfiguration.

Le Metayer [Le Metayer 1998] uses graph grammars to formalise architectures and architectural styles. A grammar defines a language of graphs, and elements of the language so defined are architectures in that style. The "coordinator" part of the grammar is devoted to reconfiguring the architecture as it evolves over time (for instance, as clients join and leave the system). Degano and Montanari [Degano and Montanari 1987] also deal with dynamic reconfiguration of architectures represented as hypergraphs.

Baresi et al. [Baresi, et al. 2004] deal with style based refinement of dynamic software architectures. The state of a system is represented as a graph while graph transformations represent state transitions. Style is formalized through type graphs, and style refinement is effected by partial surjective graph homomorphisms between the refined and the abstract type graphs, meaning that some architectural elements are deleted under the transformation. This notion of refinement differs from the one in this paper where every component of the refined architecture lies 'within' some component of the abstract architecture.

In many instances a CBS and its architecture can be considered as a complex system and analysed with the same, general machinery as biological, physical and sociological systems. A treatment of complex systems which is intuitively close to the present perspective is given by Ehresmann and Vanbremeersch [Ehresmann and Vanbremeersch 1987] who treat systems as categories, their components as objects and their connections as arrows. Evolution of systems is then, roughly, a functorial relationship. Abstraction of a subsystem is its limit, but the conditions (*clustering*) under which connections between subsystems are reflected at the level of their limits is rather more restrictive than the notion in common usage by system architects.

1.5.3 Generic Architectural Transformations

Erdogmus [Erdogmus 1998] formalises box-and-line diagrams using a set-theoretic model and presents a comprehensive set of transformations of these diagrams, which encompass virtually all the ways an architecture might be transformed. However, the set-theoretic definition of the transformations is cumbersome compared to the graph based definition presented in this paper. Further, there is no explicit interpretation of these transformations as architectural refinement for the purposes of design.

1.5.4 Abstraction and Refinement

Fahmy and Holt [Fahmy and Holt 2000a, Fahmy and Holt 2000b] model architecture as a graph. Graph rewriting is used to transform the architectures in a variety of situations: architectural understanding, analysis and modification, however architectural refinement is not explicitly modelled for the purposes of design. The notion of abstraction and refinement is lifting (showing) and hiding various existing elements of the architecture, in contrast with the present treatment where abstraction is used to encapsulate detail.

Medvidovic and Taylor [Medvidovic and Taylor 2000] present a survey and comparison of ten different ADLs. Each ADL has a different focus, ranging from modelling of dynamic behaviour and deadlock detection, to simulation of dynamic behaviour, to ADLs for certain types and domains of systems, to refinement. Of these, only SADL [Moriconi and Riemenschneider 1997] is strongly concerned with refinement. Moriconi [Moriconi, et al. 1995] models the architecture as mathematical theories, using predicates. Pre-defined and pre-proven rewriting "patterns" are proposed that can be reused at will by the engineer. However, the patterns illustrated are very low level (such as turning a pipes and filter pattern into a shared variable in code) and the research needs to build on this to make higher-level patterns that can be used freely at an architectural level of abstraction. In addition, the use of predicate-based mathematical theories results in an architectural model and refinement method that is not intuitive or practical.

Bolusset and Oquendo [Bolusset and Oquendo 2002] use rewriting logic to model and perform refinement. An example of a simple refinement (replacing a connection with a shared resource) using rewriting logic requires 21 rules and 33 equations. It is evident that the number of rules involved in a more complex refinement would grow quickly, making the method difficult to scale.

[Section 2] explicitly discusses the architectural concepts to be modelled. The mathematics of poset-labelled graphs is presented in [Section 3], and an explanation of how it can be used to model architecture and refinement follows in [Section 4]. We conclude in [Section 5] with an evaluation of the method and an indication of future developments in [Section 6].

2 Architectural Concepts to be Modelled

Several characteristics of the architecture of CBSs need to be modelled in order to develop a formalism that can be of use to the practitioner.

Firstly, any concept of architecture always includes the structure of the system, and structure primarily consists of the topology of the system. That is, the ability to specify that certain components connect to each other in certain ways. Therefore, the formalism must model topology.

Secondly, typing of components and connections is what distinguishes an architectural model from a graph – it defines the correspondence between a component or connection and some real world entity. Further, it is clear that topology alone is not sufficient to uniquely identify architectures. Two architectures may be topologically identical but represent vastly different systems. For example one star topology architecture might represent a print server being used by multiple print

clients, while another might represent the centralised control of several factory robots in a manufacturing plant. Typing of components and connections can distinguish these architectures. As such, typing should be modelled by the formalism.

The successful design of CBSs will be aided by the ability to successively refine a high level abstract architecture into a low level concrete architecture. That is, to take an architecture at a high level of abstraction (general), and refine it into an architecture at a low level of abstraction (detailed). This is an idea well understood in computer science and engineering [Abrial, et al. 1979, Bass, et al. 2003, Rechtin 1991, Ward and Mellor 1985]. This implies that the formalism must model the concept of architectures existing at differing levels of abstraction, and given that an architecture consists of components and connections, this means the typing on components and connections must exist at different levels of abstraction.

Finally, it has been proposed that refinement is the converse of abstraction [Miller, et al. 2001] and therefore by having a formal definition of architectural abstraction one can thus refine. Accordingly, if the formalism supports the ability to actually abstract an architecture (take a lower level, more concrete architecture, and abstract it into a higher level, more abstract architecture) then it will also support the ability to refine.

Therefore, the following concepts related to the architecture of CBSs are to be modelled by the formalism presented in the following section: *topology*, *type*, *levels of abstraction for types*, and *architectural abstraction*.

3 A Formal Definition of Architecture with Types and Abstraction in the Category Poset-Labelled Graphs

Poset labelled graphs were first studied in [Parisi-Presicce, et al. 1986] and they are used as the formal basis of the model.

Fix posets Π and Λ. A *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 *vertices* and *edges* respectively; $s_G, t_G : E_G \to V_G$ define the *source* and *target* of an edge; and $\pi_G : V_G \to \Pi$ and $\lambda_G : E_G \to \Lambda$ are labels of the vertices and edges.

A morphism $\phi : G \to H$ of poset labelled graphs is a pair $(\phi_V : V_G \to V_H, \phi_E : E_G \to E_H)$ such that for all $x \in V_G, e \in E_G : s_H(\phi_E(e)) = \phi_V(s_G(e))$ and $t_H(\phi_E(e)) = \phi_V(t_G(e))$ meaning ϕ preserves the structure of the graphs; and such that $\pi_H(\phi_V(x)) \geq \pi_G(x)$ and $\lambda_H(\phi_E(e)) \geq \lambda_G(e)$ which is to say that ϕ has *lax* preservation of the labelling.

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 so that poset labelled graphs and their morphisms form a category, which is denoted by $\mathrm{Graph}_{\Pi,\Lambda}$.

It is obvious that:

Proposition 1. *The epis of* $\mathrm{Graph}_{\Pi,\Lambda}$ *are precisely the arrows* ϕ *for which* ϕ_V *and* ϕ_E *are both onto, while the monos are the arrows for which they are both injective.*

Define an *embedding* to be a monomorphism $i: G \to H$ which *strictly* preserves labels $(\pi_H(i_V(x)) = \pi_G(x)$ and $\lambda_H(i_E(e)) = \lambda_G(e))$. It is routine to show that embeddings are precisely the regular monos of $\mathrm{Graph}_{\Pi,\Lambda}$.

3.1 Transforming Poset Labelled Graphs

Let $\alpha: L \to R$ be any arrow of $\mathrm{Graph}_{\Pi,\Lambda}$ and let $i: L \to G$ be an embedding. Then it is a consequence of [Parisi-Presicce, et al. 1986] Lemma 3.9 that the pushout

$$\begin{array}{ccc} L & \xrightarrow{\alpha} & R \\ {\scriptstyle i}\downarrow & & \downarrow{\scriptstyle j} \\ G & \xrightarrow[\varphi]{} & H \end{array}$$

Figure 1: Poset labelled graph pushout

exists and that H is the graph obtained by deleting L from G and replacing it with R, precisely: $V_H = V_G \setminus i(V_L) \cup V_R$ and $E_H = E_G \setminus i(E_L) \cup E_R$, while

$$s_H(e) = \begin{cases} s_R(e) \in V_R \text{ if } e \in E_R \\ s_G(e) \in V_G \text{ if } s_G(e) \notin i(V_L) \\ \phi(i^{-1} s_G(e)) \text{ otherwise } (s_G(e) \in i(V_L)) \end{cases}$$

Define t_G similarly. For vertices $x \in V_H$ define

$$\pi_H(x) = \begin{cases} \pi_G(x) \text{ if } x \in V_G \setminus i(V_L) \\ \pi_R(x) \text{ otherwise} \end{cases}$$

and λ_H similarly. Let j be the obvious inclusion and define φ on vertices (and on edges similarly) by

$$\varphi_V(x) = \begin{cases} x \text{ if } x \in V_G \setminus i(V_L) \\ \phi(i^{-1}(x)) \text{ otherwise.} \end{cases}$$

We then refer to H as the *transformation* of G by α with embedding i and write $G \overset{\alpha,i}{\Rightarrow} H$. This is a special case of *double-pushout graph rewriting* [Parisi-Presicce, et al. 1986] which also permits for parts of G to be deleted and not replaced by any part of R. Therefore graph rewriting has the potential to model more general transformations of architectures than just abstraction, but this is beyond the scope of the present work.

3.2 Architectures and Abstraction

For the purposes of our formal model, we define an *architecture* to be an object of $\mathrm{Graph}_{\Pi,\Lambda}$. The poset Π is the set of *component types* and the poset Λ contains the *connection types*. If $t \le t' \in \Pi$ (or Λ) we say that the type t' is an *abstraction* of the type t.

Let C be a distinguished set of epimorphisms of $\mathrm{Graph}_{\Pi,\Lambda}$ and refer to its elements as *abstraction rules*. Given an abstraction rule $\alpha : L \to R$ and an embedding $i : L \to G$ as in [Fig. 1], then the transformation $G \overset{\alpha,i}{\Rightarrow} H$ is an *architectural abstraction*, and we say that *H* is an *abstraction* of *G* and that *G* is a *refinement* of *H*. Since pushouts preserve epimorphisms, we can see that any $\varphi : G \to H$ arising in this way is again an epimorphism. Therefore we can regard abstraction transformations as special epimorphisms respecting structure as defined by the abstraction rules. We will also refer to a sequence $G \overset{\phi_1,i_1}{\Rightarrow} G_1 \overset{\phi_2,i_2}{\Rightarrow} \ldots \overset{\phi_k,i_k}{\Rightarrow} H$ of abstractions as an *architectural abstraction*.

3.3 How the Formalism is Coupled to the Real-World

In [Section 2] it was stated that the following architectural concepts should be modelled: *topology*, *type*, *levels of abstraction for types*, and *architectural abstraction*.

As previously stated, the underlying principle is that the entire formalism should be tightly coupled with "real world" aspects of the architectural design process such that it is immediately obvious what any of the mathematical constructs are intended to model.

Topology is modelled by the graphs themselves. A graph consists of vertices, which model the components, and edges, which model the connections. *Type* is modelled by the labels on the graphs. *Levels of abstraction for types* are modelled by the fact that the labels (types) are elements of a poset, which is ordered according to levels of abstraction, and *architectural abstraction* is modelled explicitly by the abstraction rules. It is hoped that this correspondence between the formalism and the architectural constructs is clear, unambiguous and easy to use.

3.4 Technical Features of the Model

Once appropriate component types, connection types and abstraction rules are selected for a particular domain, the definitions above constrain the notions of architectural abstraction and refinement in useful ways. Consider the following simple example from the domain of 'office automation' with component and connection type hierarchies (posets) as depicted below:

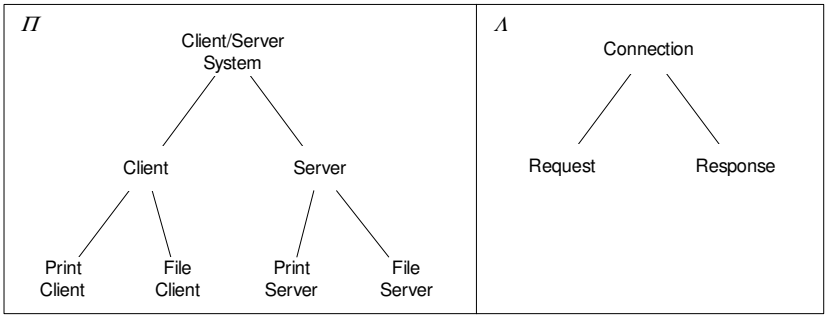

Figure 2:Client / Server system posets

together with the following (infinite) abstraction rule sets:

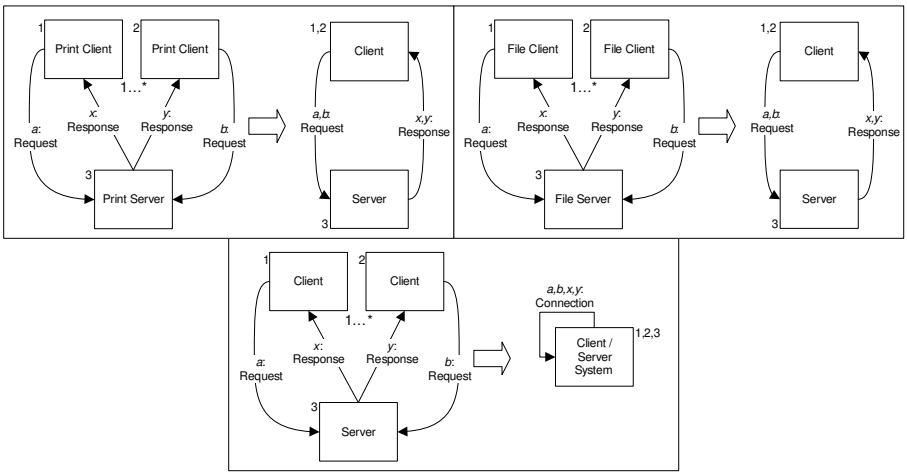

Figure 3: Abstraction rules for Client/Server System

Now we are able to illustrate the following features of our formal system.

3.4.1 Architectural Abstraction is Stronger than Type Abstraction

Abstracting only types, one could start with the architecture:

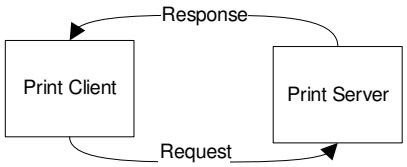

Figure 4: A valid Print-Client / Print-Server Architecture

and abstracting only the type of the 'Print Client' component, arrive at

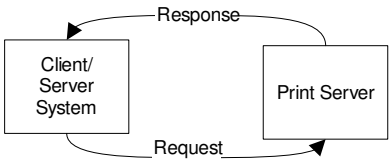

Figure 5: Invalid abstraction of Print-Client / Print-Server

which is clearly an inaccurate model of the application originally described. This model cannot arise as an abstraction of the original architecture through application of the abstraction rules above.

3.4.2 Architectural Abstraction is Stronger than Epimorphism

If we were to admit any epimorphism between poset labelled graphs as an architectural abstraction, then

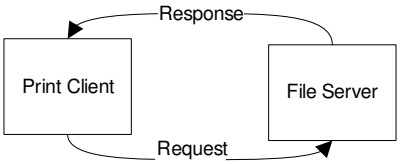

Figure 6: Invalid refinement of Client / Server System

would be a refinement of

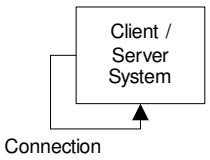

Figure 7: Client / Server System

However using the abstraction rules selected for this domain, no such absurd architecture may arise as a refinement of client-server system.

3.4.3 Architectural Refinement as a Faithful Interpretation of Architectural Theories

In [Moriconi, et al. 1995] architectures are considered as logical theories, and a refinement is a faithful interpretation from the abstract theory into the concrete. This encodes two conditions that one would wish to impose on any reasonable notion of refinement, namely:

- The fact that refinement is an *interpretation* means that if a sentence is true of the abstract architecture, then its interpretation is true of the refined architecture.
- The *faithfulness* condition means that if a sentence is not true of the abstract architecture, then its interpretation is not true of the concrete architecture.

Since the sentences in [Moriconi, et al. 1995] refer to connectedness of components, these conditions translate to the present graph-theoretic setting as:

- *Homomorphism*: If two components are connected in the refined architecture, then their abstractions are connected in the abstract architecture. This is the contrapositive of the faithfulness condition.
- *Epimorphism*: If two components are connected in the abstract architecture, then there is some direct connection between components of the subsystems they represent. This corresponds to refinement being an interpretation of theories.

4 Usage of the Model

This section illustrates the usage of the formalism presented in [Section 3]. The usage scenario shows the tasks involved for a team entrusted with the task of designing a customer relationship management (CRM) system for use within a customer facing organisation. As the section proceeds, the story will be further narrated.

There are two distinct tasks that must be performed in order to use the poset labelled graph formalism. The first is the task of developing: the posets of types for components and connections (Π and Λ respectively, from [Section 3]); the set of abstraction rules (C from [Section 3.2]); and, an initial abstract architecture of the system. Once this is done, the second task is to propose refined architectures and check whether they are valid refinements of the abstract architecture.

4.1 Task 1 – Prepare Posets, Rules and Initial Abstract Architecture

Task 1 would normally be performed by one or more people with knowledge of the domain (domain experts) as it requires them to analyse and determine: the types of components and connections that will exist in the system at varying levels of abstraction; the valid abstraction rules; and a sensible initial abstract architecture.

4.1.1 Developing the Posets

Firstly, one needs to determine the types of components and connections within the domain (CRM system) and place them into posets. For the purposes of this illustration, only the component type poset (Π) will be developed. The connection type poset is assumed to exist but with only one element, 'Uses'. Accordingly, all connections in the architectures are assumed to be of type 'Uses', as in "component *x uses* component *y*".

As previously stated, the main objective of the design activity is to design a CRM system for use within an organisation with customers. The domain experts begin their analysis of the domain and realise that the overall *'System'* would consist of *'Customers'*, *'Internal Reports Departments'* (i.e. people within the organisation who

wish to use and gather information from the CRM system) and the *'CRM System'* itself. Already four component types have been identified.

Further, the domain experts realise that, given the size of the organisation, there may be more than one *'Internal Reports Department'* that would wish to use the CRM System. Therefore they create another type, *'Internal Reports Department'* and designate it as a refinement of *'Internal Reports Departments'* (note the singular and the plural, representing different component types).

Understanding the likely component types that would be used to construct a CRM System, the domain experts decide that the CRM system could be modelled as some kind of component that would store and routinely manage the CRM data, and they designate this component type *'Data Storage and Processing'*. To facilitate the use of the system (by customers and internal reports departments) the experts decide that some kind of component is needed that allows the users to interact with the system. They designate these components to be of type *'Online Application'*.

Finally, the domain experts decide that a *'Data Storage and Processing component'* would in reality consist of *'Database'* components (to store the data) and *'Batch Application'* components (to manipulate and manage the data).

The result of this analysis is that a Hasse diagram for the poset can be drawn:

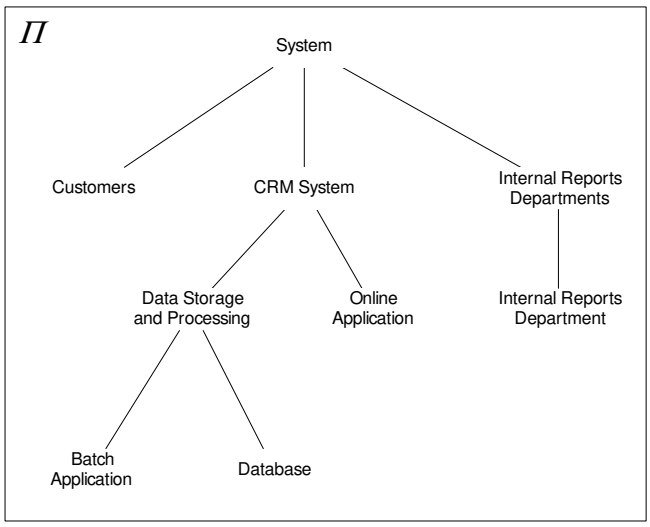

Figure 8: Hasse diagram of componenet types

It is important to note that this structure does *not* represent an architecture. It is a diagrammatic representation of the poset of component types (Π).

4.1.2 Developing the Abstraction Rules

Now that the posets are established, the domain experts can begin to formulate the set of abstraction rules (C). These rules are shown in the following figure.

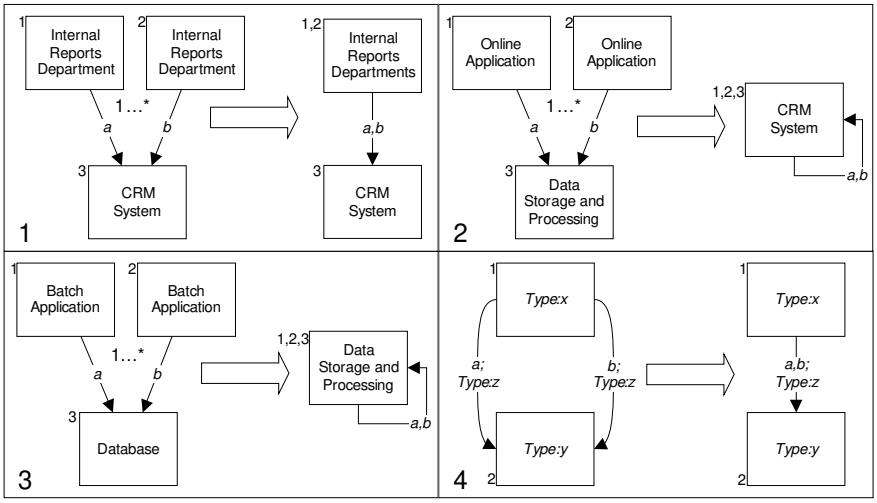

Figure 9: Abstraction rules

In the above figure, the "1…*" notation indicates an infinite set of rules. Using Rule 1 as an example, this translates to having a separate rule for 1, 2, 3 (and so on) 'Internal Reports Department' components being connected to a 'CRM System' component. This is more an issue if the formalism is automated, for example with a program like AGG [AGG 2002] where each rule would have to be entered. In practice one would simply enter enough rules with a reasonable upper bound and use a "pre-processor" to do this.

Rule 4 represents a meta-rule, where *x*, *y* and *a* are variables. The rule can be instantiated with any component types for *x* and *y*, and any connection type for *a*. The rule essentially states that any two components connected by two connections of the same type, in the same direction, may be abstracted to those two components connected by only one connection. The use of Rule 4 is illustrated in [Section 4.2.2].

The rationale for developing Rule 1 is that 'Internal Reports Department' is a refinement of 'Internal Reports Departments' and that both would connect to (i.e. use) the 'CRM System'.

The rationale for developing Rule 2 was hinted at in [Section 4.1.1] when the types were created, when the domain experts realised that a 'CRM System' could be modelled as an 'Online Application' connected to a 'Data Storage and Processing' component. Further, multiple 'Online Application' components may be connected to a 'Data Storage and Processing' component to facilitate different types of interactions with the 'Data Storage and Processing' component, for example: updating customer details versus reporting on customer demographics.

The rationale for developing Rule 3 is similar to that of Rule 2, except this time realising that a 'Data Storage and Processing' component could be modelled as a 'Database' connected to multiple 'Batch Application' components, to facilitate different types of data manipulation and processing, for example a month end processing of customer loyalty points versus processing to calculate demographic data.

4.1.3 Developing the Initial Abstract Architecture

Finally, before the designer may begin to propose architectures, an initial abstract architecture must be developed such that the designer knows the starting point. The initial abstract architecture is hinted at during the domain analysis of [Section 4.1.1], and is as follows:

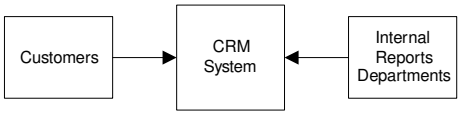

Figure 10: Initial abstract architecture

Note that each component in our architectures may be taken to have a self-connection (loop) of type 'Uses' which will be omitted from the diagrams. This connection will be the image of any connections within a subsystem which is abstracted to the component.

4.2 Task 2 – Propose Refined Architectures and Check

Task 2 would normally be performed by a person (the designer) with knowledge of the types available in the domain. This person may or may not be one of the domain experts, and if they are not, then the types can be communicated by the posets developed in Task 1.

4.2.1 Proposing Refined Architectures

This activity must assume some basic requirements or expectations of the system (full requirements analysis and specification is outside the scope of this work and this illustration), as well as the initial abstract architecture (see [Fig. 10]). For this usage scenario, the following basic requirements are given to the designer:

- There are two internal reports departments who wish to use the system. The first must be able to view reports on customer demographics, and the second must also view these reports but also be able to generate quarterly letters to be sent to customers regarding their "customer loyalty scheme" points;
- Customers must be able to update their own details;
- Customers must be able to view their "customer loyalty scheme" points;
- The system must be able to calculate demographic trends based on the stored CRM data.

Based on these, the designer may propose the following architecture as a valid refinement, or design, of the system to fulfil all requirements.

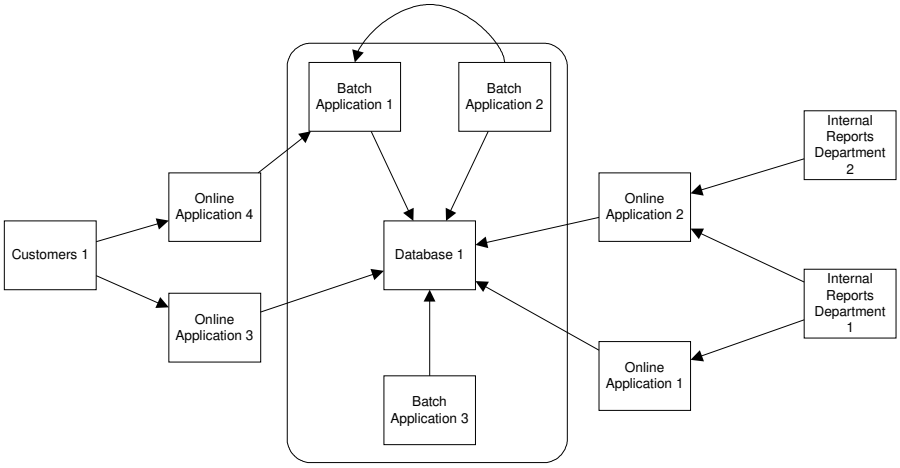

Figure 11: Proposed refined architecture

The notation used in the architectures is "<component type> <number>". This notation is used firstly to identify the type of each component, and secondly to differentiate different instances of components of a particular type. Note that the meaning of the rounded box around the 'Batch Application' and 'Database' components is explained in the following section.

The designer provides the following rationale behind this design:

- Database 1 stores all the CRM data;
- Online Application 1 generates the customer loyalty letters;
- Online Application 2 reports on customer demographics;
- Online Application 3 allows customers to view their loyalty scheme points;
- Online Application 4 allows customers to update their details;
- Batch Application 1 receives the customer updates and accordingly updates the database. It is also used by Batch Application 2, which receives the customer updates, calculates related demographic data, and updates the database.
- Batch Application 3 performs other trending and analysis on the customer demographic data in the database.

4.2.2 Checking if the Refined Architectures are Valid

The final step is to apply the abstraction rules to the refined architecture to determine whether or not it can be transformed back into the abstract architecture, i.e. whether or not it is a valid refinement.

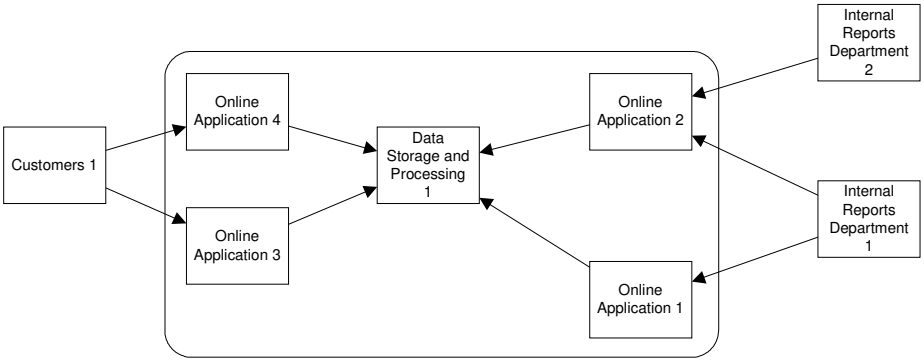

Figure 12: Architecture after applying Rule 3

In the Figure above, Rule 3 has been applied to the refined architecture in [Fig. 11]. The circled components in [Fig. 11] are the components that are matched by the left-hand side of Rule 3 such that the 'Batch Application' components connected to the 'Database' are replaced by 'Data Storage and Processing 1', producing [Fig. 12].

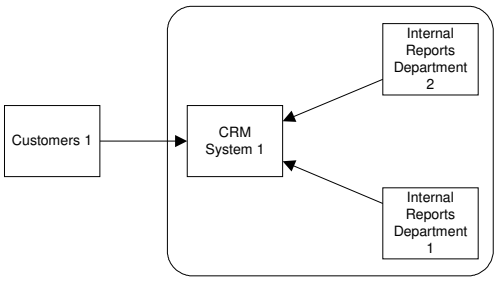

Figure 13: Architecture after applying Rule 2

In the Figure above, Rule 2 has been applied to the architecture shown in [Fig. 12]. The circled components in [Fig. 12] are the components that are matched by the left-hand side of Rule 2 such that the 'Online Application' components connected to the 'Data Storage and Processing' component are replaced by 'CRM System 1', producing [Fig. 13]. It should be noted at this point that when Rule 2 is applied to the architecture in [Fig. 12] *two* connections exist between 'Internal Reports Department 1' and 'CRM System 1' and there are also *two* connections between 'Customers 1' and 'CRM System 1'.

This sort of situation is likely to arise often, and is handled by Rule 4. The rule would be instantiated twice. Once with *x* replaced by 'Internal Reports Department' and *y* replaced by 'CRM System', and once with *x* replaced by 'Customers' and *y* replaced by 'CRM System'. In both cases *a* is replaced by 'Uses' as this is the only connection type in this example. These two instances of Rule 4 are assumed to have been applied to the architecture in [Fig. 12] in order to eliminate the multiple connections and produce the architecture in [Fig. 13].

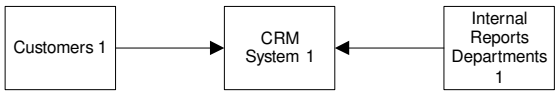

Figure 14: Architecture after applying Rule 1

Finally, in the above figure, Rule 1 has been applied to the architecture shown in [Fig. 13]. The circled components in [Fig. 13] are the components that are matched by the left-hand side of Rule 1 such that the two 'Internal Reports Department' components connected to the 'CRM System' are replaced by one 'Internal Reports Departments 1' component connected to 'CRM System 1', producing [Fig. 14].

By comparing the architecture in [Fig. 14] to the initial proposed abstract architecture in [Fig. 10], and bearing in mind the "<component type> <number>" notation, it is evident that the two architectures are the same and therefore it can be concluded that the proposed refined architecture of [Fig. 11] is indeed a valid refinement.

This concludes the illustration of how the formalism can be set up and used in a real-world situation.

5 Evaluation

The formal framework presented is an *architectural modelling language* and a *calculus* for transforming between models of the same system at different levels of abstraction. This framework is intended to have the following properties:

- *rigorous*: using the calculus ensures models at different levels of abstraction are coherent as descriptions of a system (see [Section 3.4.3])
- *expressive*: knowledge of the application domain can be encoded in the calculus to constrain transformations.
- *simple*: the formal concepts required to use the framework are within the reach of most practitioners
- *practical*: lightweight enough to be applied without any special tools and without being disproportionately time consuming.

We now evaluate the present work in terms of these criteria.

5.1 Simplicity

Architecture for our purposes is taken to mean *type* (of components and connections) and *topology* (which components are directly connected, and in what way). Types of components and connections are arranged in hierarchies, and are used to establish a semantic mapping into the "real-world'. It models refinement as decomposition of components into subsystems, and abstraction as encapsulating subsystems into components. This is in contrast with the Fahmy and Holt notion of abstraction as information hiding [Fahmy and Holt 2000a, Fahmy and Holt 2000b], which is simple, but less useful for dealing with large systems. The mathematical definitions associated with the category of poset labelled graphs are compact, and intuitive with a certain amount of practice – most of the authors are non-mathematicians.

5.2 Practicality

Like [Erdogmus 1998] and [Fahmy and Holt 2000a, Fahmy and Holt 2000b] our formalism is graphical, making it quick and intuitive to read and write as the reader will readily see from [Section 4], and as some of the authors can attest, from use within their consulting work. This is in marked contrast with much of the work in this area, for instance Moriconi [Moriconi, et al. 1995], where ADLs are first order predicate languages, which impose an unreasonable burden of time and concentration on the reader to understand.

Unlike Erdogmus [Erdogmus 1998] and Le Metayer [Le Metayer 1998], the transformations used are based on the simple concept of a graph pushout which can easily be applied by hand (although these computations can certainly be performed using a graph calculator such as AGG [AGG 2002]).

It is important to note that the transformations used in this paper all act on the architecture locally, unlike when type-graphs are involved [Baresi, et al. 2004] where everything of a given type will be transformed simultaneously. This means that large architectures may be manipulated by focussing only on the subsystems of interest.

5.3 Rigor and Expressiveness

[Section 3.4] illustrates how rules can be used to express domain knowledge as constraints on transformations, and also shows that defining abstraction as graph epimorphism ensures coherence of the models at different levels of abstraction. This is in contrast with previous graph transformation based approaches of Fahmy and Holt [Fahmy and Holt 2000a, Fahmy and Holt 2000b] and Erdogmus [Erdogmus 1998] which do not seek to ensure that any particular properties are maintained by the transformations.

6 Future work

Several desirable features of a formal framework have not been incorporated into the present work. Specifically:

- There is no way of expressing rules which constrain the form of an architectural model. Within the present framework it is quite possible to write down an architecture in which a print-client makes requests to a file-server instead of a print server (see [Section 3.4]), although the present formalism will determine that such an architecture is not a valid refinement of client/server system.
- There is no way of ensuring coherence of the level of abstraction across an architecture – if you abstract one (print-client $\rightarrow$ print-server) pair into a client-server system, you should abstract all the print-clients connected to that print server.

Future work addressing these issues will necessarily be directed toward an automated system so that these added constraints are maintained by a program, and a designer may:

- request the application of an abstraction rule to an architecture, and allow the program to compute the abstracted architecture;

- propose an abstraction relationship between two architectures and allow the computer to determine whether (possibly multiple) applications of abstraction rules suffice to derive one architecture from the other.

As this work goes forward, the major challenge will be to formalize and automate architectural transformation in a way which will guide and reinforce the engineer's intuition.

Acknowledgements

Funding for this work was furnished in part by the Australian Research Council (ARC) in the form of an Australian Postgraduate Award (APA) scholarship for Mark Denford, and in part from Avolution Pty. Ltd. as part of a joint project. The second author acknowledges helpful discussions with Dr. Steve Lack, and also with Professor Hartmut Ehrig, Olga Runge and Dr. Gabriele Taentzer while on sabbatical at the Technische Universität Berlin.

References

[Abrial, et al. 1979] Abrial, J. R., Schuman, S. A., and Meyer, B.: "Specification Language Z"; Massachusetts Computer Associates Inc., Boston (1979)

[AGG 2002] AGG, "The Attributed Graph Grammar System, http://tfs.cs.tu-berlin.de/agg/": Institut für Softwaretechnik und Theoretische Informatik, Fakultät IV - Elektrotechnik und Informatik, Technische Universität Berlin, (2002).

[Alexander 1964] Alexander, C.: "Notes on the synthesis of form"; Harvard University Press, Cambridge (1964)

[Allen 1997] Allen, R.: "A Formal Approach to Software Architecture", PhD thesis, Carnegie Mellon, School of Computer Science, 1997.

[Avolution 2004] Avolution, "ABACUS - Architecture Based Analysis of Complex Systems, http://www.avolution.com.au", (2004).

[Baresi, et al. 2004] Baresi, L., Heckel, R., et al.: "Style-based refinement of dynamic software architectures"; Proc. Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on, (2004), 155-164.

[Bass, et al. 2003] Bass, L., Clements, P., and Kazman, R.: "Software architecture in practice"; Addison-Wesley, Reading, Mass. (2003)

[Bolusset and Oquendo 2002] Bolusset, T. and Oquendo, F.: "Formal Refinement of Software Architectures Based on Rewriting Logic"; Proc. International Workshop on Refinement of Critical Systems: Methods, Tools and Experience, Gronoble, France (2002)

[Degano and Montanari 1987] Degano, P. and Montanari, U.: "A model for distributed systems based on graph rewriting"; J. ACM, 34, 2 (1987), 411-449.

[Denford, et al. 2003] Denford, M., O'Neill, T., and Leaney, J.: "Architecture-based design of computer based systems"; Proc. Engineering of Computer-Based Systems, 2003. Proceedings. 10th IEEE International Conference and Workshop on the, Huntsville, Alabama, USA (2003), 39-46.

[Ehresmann and Vanbremeersch 1987] Ehresmann, A. C. and Vanbremeersch, J.-P.: "Hierarchical Evolutive Systems: A mathematical model for complex systems"; Bulletin of Mathematical Biology, 49, 1 (1987), 13-50.

[Erdogmus 1998] Erdogmus, H.: "Representing Architectural Evolution"; Proc. Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research, IBM Press, Toronto, Ontario, Canada (1998), 159-177.

[Fahmy and Holt 2000a] Fahmy, H. and Holt, R. C.: "Software architecture transformations"; Proc. Proceedings. International Conference on Software Maintenance, 2000., (2000a), 88-96.

[Fahmy and Holt 2000b] Fahmy, H. and Holt, R. C.: "Using graph rewriting to specify software architectural transformations"; Proc. The Fifteenth IEEE International Conference on Automated Software Engineering, 2000. Proceedings ASE 2000., (2000b), 187-196.

[Guttag, et al. 1993] Guttag, J. V., Horning, J. J., et al.: "Larch: Languages and Tools for Formal Specification"; Springer-Verlag, New York, NY (1993)

[Hoare 1985] Hoare, C. A. R.: "Communicating sequential processes"; Prentice/Hall International, Englewood Cliffs, N.J. (1985)

[Horowitz 1991] Horowitz, B. M., "The importance of architecture in DOD Software," MITRE Corporation, Bedford, Massachusetts, USA M91-35, 1991.

[IEEE 2000] IEEE, "IEEE Std 1471-2000: IEEE Recommended practice for architectural description of software-intensive systems", (2000).

[Jones 1980] Jones, C. B.: "Software development : a rigorous approach"; Prentice/Hall International, Englewood Cliffs, N.J. (1980)

[Lavi 1991] Lavi, J.: "Formal establishment of computer-based systems engineering field urged"; IEEE Computer, 24, 3 (1991), 105-107.

[Lawson, et al. 1999] Lawson, H. W., Leaney, J., and O'Neill, T.: "Open complex computer based systems: only the first step along the way to safe, reliable computing"; Proc. Engineering of Computer-Based Systems, 1999. Proceedings. ECBS '99. IEEE Conference and Workshop on, Application, (1999), 294-301.

[Le Metayer 1998] Le Metayer, D.: "Describing software architecture styles using graph grammars"; Ieee Transactions on Software Engineering, 24, 7 (1998), 521-533.

[McMenamin and Palmer 1984] McMenamin, S. M. and Palmer, J. F.: "Essential systems analysis"; Yourdon Press, Englewood Cliffs (N.J.) (1984)

[Medvidovic and Taylor 2000] Medvidovic, N. and Taylor, R. N.: "A classification and comparison framework for software architecture description languages"; Software Engineering, IEEE Transactions on, 26, 1 (2000), 70-93.

[Miller, et al. 2001] Miller, J., Mukerji, J., et al., "Model Driven Architecture (MDA)," Object Management Group, ORSMC/2001-07-01, 2001.

[Milner 1993] Milner, R., "The polyadic pi-calculus: a tutorial," in Logic and Algebra of Specification, F. L. Bauer, W. Brauer, and H. Schwichtenberg, Eds.: Springer-Verlag, 1993, pp. 203--246.

[Moriconi, et al. 1995] Moriconi, M., Qian, X., and Riemenschneider, R. A.: "Correct architecture refinement"; Software Engineering, IEEE Transactions on, 21, 4 (1995), 356-372.

[Moriconi and Riemenschneider 1997] Moriconi, M. and Riemenschneider, R. A., "Introduction to SADL 1.0: A Language for Specifying Software Architecture Hierarchies," SRI International, SRI-CSL-97-01, 1997.

[Neumann 2004] Neumann, P. G., "Forum On Risks To The Public In Computers And Related Systems (comp.risks) http://www.csl.sri.com/users/risko/risks.txt", vol. 2004: ACM Committee on Computers and Public Policy, (2004).

[O'Neill, et al. 2000] O'Neill, T., Leaney, J., and Martyn, P.: "Architecture-based performance analysis of the COLLINS class submarine open system extension (COSE) concept demonstrator (CD)"; Proc. Engineering of Computer Based Systems, 2000. (ECBS 2000) Proceedings. Seventh IEEE International Conference and Workshopon the, Practical, (2000), 26-35.

[Parisi-Presicce, et al. 1986] Parisi-Presicce, F., Ehrig, H., and Montanari, U.: "Graph rewriting with unification and composition. Graph-grammars and their application to computer science"; Lecture Notes in Computer Science, (1986), 496-514.

[Payne 1999] Payne, C. N.: "Using Composition & Refinement to Support Security Architecture Trade-off Analysis"; Proc. 22nd National Information Systems Security Conference, NIST, Virginia, USA (1999)

[Rechtin 1991] Rechtin, E.: "Systems architecting : creating and building complex systems"; Prentice Hall, Englewood Cliffs, N.J. (1991)

[Rowe 1999] Rowe, D.: "An Ontological Model of Architectural Change for Computer Based Systems", Doctoral thesis, University of Technology, Sydney, Sydney1999.

[Rowe and Leaney 1997] Rowe, D. and Leaney, J.: "Evaluating evolvability of computer based systems architectures-an ontological approach"; Proc. Engineering of Computer-Based Systems, 1997. Proceedings., International Conference and Workshop on, (1997), 360-367.

[Shaw and Garlan 1996] Shaw, M. and Garlan, D.: "Software architecture : perspectives on an emerging discipline"; Prentice Hall, Upper Saddle River, N.J. (1996)

[Sommerville 2001] Sommerville, I.: "Software engineering"; Addison-Wesley, Harlow, England ; New York (2001)

[van Lamsweerde 2000] van Lamsweerde, A.: "Formal specification: a roadmap"; Proc. Conference on The future of Software engineering, ACM Press, Limerick, Ireland (2000), 147-159.

[Ward and Mellor 1985] Ward, P. T. and Mellor, S. J.: "Structured development for real-time systems"; Yourdon Press, New York, N.Y. (1985)

[Wordsworth 1991] Wordsworth, J. B.: "The CICS application programming interface definition"; Proc. Z User Workshop, Springer-Verlag, Oxford (1991)