Architecture-Based Design of Computer Based Systems

Mark Denford, Tim O'Neill, John Leaney
Architecture-based Engineering Research Program
Institute for Information and Communication Technologies
University of Technology, Sydney
[Mark.Denford | Tim.ONeill | John.Leaney]@uts.edu.au

Abstract

This paper presents a practical approach to architecture-based design of computer based systems. The approach is discussed in relation to other existing methods of performing discovery, abstraction, refinement and evolution of systems' architectures. It has also be shown that this approach can be supported by formal methods of refinement. The approach assists the designer to maintain a strict focus of reasoning about the architecture and its qualities.

1. Introduction

The importance of architecture in the engineering of computer based systems is widely recognised [19, 22, 34, 38]. Given a strong architectural model [30], the architectures of systems can be visualised [9], reasoned about [4, 8, 27, 32], and evolved [36]. These activities are part of any process of architecture-based design, which we call A-Design. This paper particularly investigates the area of (formal) design traditionally called refinement.[1]

It is important to consider architecture-based design in a practical engineering context and in particular to address the practical concerns of the engineering effort involved in developing (short term change) and evolving [36] (long term change) a given Computer Based System (CBS). One must always be sure that in designing a system the non-functional requirements of the system are satisfied [5, 11]. Additionally, no system is ever built from nothing; in practice the designers will have suggestions for the system at every level of abstraction [3, 29, 41].

The latest IEEE Requirements standard recommends that requirements are hierarchical [21] and always have some associated implementation restrictions. As a result there will be a need to place these restrictions at whichever level of abstraction is most appropriate [5, 11, 31]. In addition to this need to be able to interact at which ever level of abstraction is most appropriate, the practicing designer will often as a first step need to discover the architecture of an as-built system. That is, they will need to develop a concrete architecture of a system and abstract away details until the underlying architecture of the system is exposed [14, 24].

This paper presents a practical approach to architecture-based design of CBSs. The approach is discussed in relation to other existing methods of performing discovery, abstraction, refinement and evolution of systems' architectures. It will also be shown that this approach (whilst generally being more practically based than other more formal methods) can be supported by these formal methods of refinement in particular. The approach also helps to maintain a strict focus of reasoning about the architecture and its qualities.

General architectural definitions are presented in section 2. Section 3 presents a discussion of related work on refinement in general and section 4 builds upon this work with our concept of architecture-based design. Finally a practical approach to architecture-based design is presented in section 5. The paper finishes with a discussion of future work and a conclusion (section 5.1).

2. Architecture

This section presents some general architectural definitions to establish the vocabulary of the paper. From the IEEE [22] and extended by the ECBS Architecture Working Group [37] and UTS [26] the following definitions are provided;

System: A set of interrelated entities which display a specified behaviour while interacting with the system's particular environment.

Architecture: Any well defined form of a system's essential, unifying structure defined in terms of components, connections and constraints along with the system's interaction with its environment.

Architectural description (A-Description): A product which documents an architecture and consists of zero or more architectural models, including rationale for and relationships between the models and views chosen.

Architectural models (A-Models): Any formal description of a system which describes the system's

[1] If refinement is the process of taking an architecture from the abstract to the concrete, then "abstracting" is the reverse process - taking an architecture from the concrete to the abstract. For the purposes of this paper we refer to both the concepts of "abstracting" and traditional refinement as (part of) design.

architecture. Typically A-Models are formulated using a specific A-Style while embodying (or portraying) one or more A-Views (refer to [26]).

Architectural Model Elements (A-Elements)*:* The constituents of a system's architectural model which represent the components (Cp), connections (Cn) and constraints (types, implementations, properties, etc) of the proposed system architecture.

Please note: For reasons of brevity and concentration, the concepts of Architectural Styles (A-Style), Architectural Patterns (A-Patterns), Architectural Principles (A-Principles) and Stakeholders concerns are extensively discussed in other places, including our working group [26] but omitted here.

In Figure 1, UML notation [40] is used as an extended entity relationship diagram, to compare and contrast the meanings of the definitions.

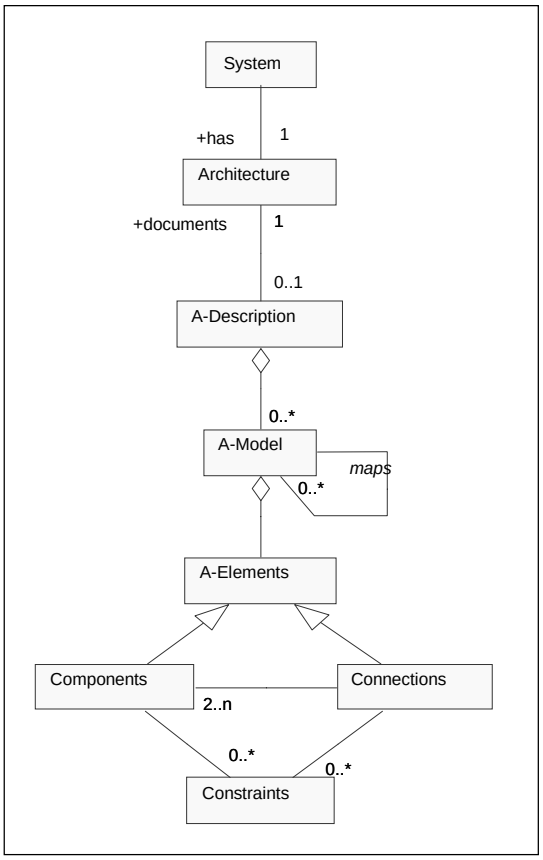

Figure 1 – Interrelationship between key Architectural terms [26]

3. Related Work

This section discusses the history of refinement methods in general from it's origins in early program proving work to the emergence of architectural refinement methods in the 1990's. Additionally, existing architectural refinement approaches are discussed, and, finally the strong relationship between requirements and refinement is argued with a recognition of the importance of hierarchical requirements to a refinement approach.

3.1 History of Refinement Methods

Existing refinement methods have their origins in the program proving work of Dijkstra [10] and Hoare [17]. The initial work on stepwise refinement by Dijkstra [10] appears to be the first to use logic in the construction of a program from a design, as specified by pre/post conditions. Refinement of programs from models gained a boost with the work on the Vienna Development Method (VDM) [23] and Z [15] and B [2]. These methods specify systems, and refine them to programs using predicate calculus and proof obligations. Z has been used on large practical systems to reverse specify and then, by proof techniques, understand and re-specify industrial systems, including the IBM CICS system [42]. They have also been used to model and prove the correctness of systems. Another approach, exemplified by the LARCH project, [25] makes use of axioms and rewriting logic to model a system and prove the correctness of an implementation.

All these methods have one major weakness in that there is no concept of design attributes being satisfied. For example, the design requirement that the system be maintainable could require the application of the principles of coupling and cohesion. One exception is Object-Z [7] which implicitly applies the principles of coupling and cohesion since it forces an information hiding paradigm on the modeller.

3.2 What is Architectural Refinement

Architectural refinement methods have been developed since the early 1990's. Broadly speaking, these methods can be classified into predicate logic reasoning and refinement methods [4], and methods focussing on rewriting logic or mapping architectural patterns and styles. [5, 31].

In [6], Büchi discusses refinement from a formal specifications perspective. He refers to refinement to be that "the implementation actually complies with its specification, or, more precisely, is a refinement thereof". This perspective is similar to the classical refinement approach which uses the notion of "behavioural substitutability". That is, the concrete representation should not show any behaviour not observable in the abstract representation [13]. This is the approach used in CSP [18]. Moriconi et al [31] argue that behavioural substitutability may not be sufficient and introduce "conservation extension". That is, if a feature is not explicitly included in the abstract then it is implicitly claimed not to exist [13]. According to the above authors, refinement is thus the process of ensuring that these

conditions of "behavioural substitutability" and "conservation extension" are met.

Garlan [13] introduces his own perspective on refinement by claiming "that there is no single definition of refinement. Rather, refinement rules must be specific about what kinds of properties they are preserving in the refined design". This sort of definition is moving towards the notion that refinement can be thought of in terms of ensuring the non-functional properties of a system such as evolvability and performance.

3.3 Refinement and Requirements

Egyed et al [11] imply a strong relationship between requirements (both functional and non-functional) and refinement. They mention the "need of having requirements engineering and architectural modeling being intertwined and mutually-dependent development activities in order to ensure their complete and consistent treatment (i.e., refinement)." This perspective is also supported by Bolusset et al [5] who state that refinement is used to ensure that the system's concrete implementation still meets its requirements.

We propose to extend Egyed et al and Bolusset et al's concepts of architectural refinement by including the concept of hierarchical requirements, supported by the latest IEEE standard on System Requirements Specifications [21]. It argues that requirements are assembled "into a hierarchy of capabilities where more general capabilities are decomposed into subordinate requirements". The implication of this to architecture-based design is that at each level of abstraction a certain subset of the overall system requirements will be addressed. This is further discussed in the following section.

4. Architecture-based Design

This section defines the architecture-based design of Computer Based Systems (CBSs). Additionally two types of architecture-based design (*horizontal* and *vertical*) are identified and discussed with regard to the differing reasons for using each one. The section concludes with a diagrammatic summary of the terms and relationships introduced in this section. This diagram provides the basis for the proposed practical design approach (section 5).

4.1 Definition

The more recent interpretations of refinement [5, 11] and the concept of hierarchical requirements [21] influences the definition of architecture-based design used within this paper. Within the context of the architecture-based design approach discussed herein and the general architectural definitions of the UTS [26] (summarised in section 2) architecture-based design (A-Design) can be defined as;

> *A-Design*: the addition (or removal) of A-Elements to the A-Model to ensure a larger subset of the overall requirements are met.

As is evident in Figure 2, A-Design can be seen in terms of developing another A-Model that satisfies more of the overall systems requirements. The initial A-Model satisfies a certain set of the system's requirements, R_1. After the refinement step, the final A-Model satisfies the set of requirements, R_2. Given that R_1 is a proper subset of R_2, the final A-Model also satisfies all of the requirements originally satisfied by the initial A-Model.

In practice, R_1 may not be a proper subset of R_2 after the first attempts at refining. The actual refinement method must detect this and ensure that the condition is met before the refinement step is considered complete. This is discussed further in section 5.

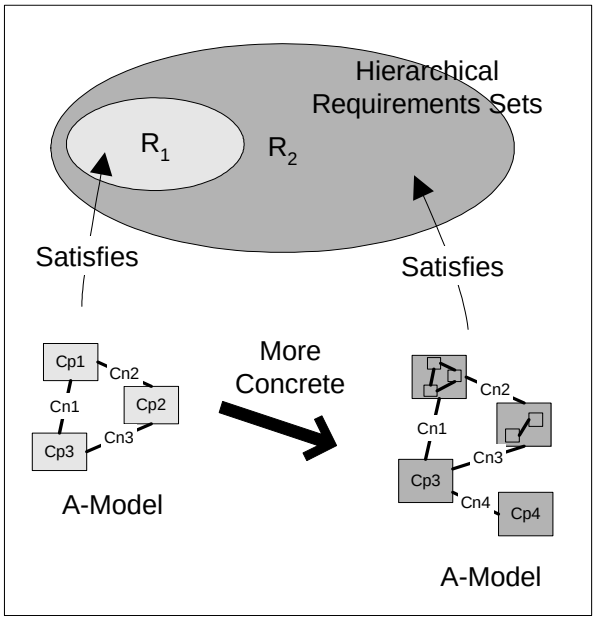

Figure 2 - Relationship between A-Design and Requirements

4.2 Types of A-Design

Following the general definition of A-Design given in section 4.1, more specific definitions can be given for two types of A-Design, *horizontal* and *vertical*, that are often discussed, though not entirely agreed upon, in literature.

Bolusset et al [5] refer to horizontal refinement as inducing a specification modification where there is no change of abstraction level. We build on this, and within our approach and architectural definitions of the UTS [26] (summarised in section 2), define;

> *Horizontal A-Design*: the addition (or removal) of A-Elements at the **same level of abstraction** to satisfy an additional subset of the overall requirements – both functional and non-functional.

In [5] the process of vertical refinement is referred to as moving "closer to the implementation by going from a first architecture description language, to a second one", that is, moving from one level of abstraction to a second one. "Details are added to remove a part of indeterminism or to facilitate implementation". This interpretation of vertical refinement as involving a transition of levels of abstraction is generally well understood.

Whilst discussing the vertical case with respect to their refinement method, Moriconi et al [31] comment that "we are guaranteed that the most concrete architecture in the hierarchy meets the requirements of the most abstract architecture in the hierarchy." Thus vertical refinement involves a transition of levels of abstraction.

We build on these concepts and within our approach and architectural definitions of the UTS [26] (summarised in section 2) define:

> *Vertical A-Design*: the addition (or removal) of A-Elements at a **more concrete or less concrete level of abstraction** to satisfy an additional subset of the overall requirements – both functional and non-functional.

Of particular note is the "more concrete or less concrete" portion of the definition which reflects the concept of abstracting (from concrete to abstract) being the reverse process of refining (from abstract to concrete).

4.3 Levels of Abstraction

A fundamental aspect of our architecture-based design (A-Design) approach is the concept of levels of abstraction for A-Models that originates from the work of Ward and Mellor. In [41] they introduce the concepts of an *Essential Model* and an *Implementation Model*:

> *"Given that a system must function in a specific environment, and given that it has a purpose to accomplish, it is possible to describe what it must do (the essential activities) and what data it must store (the essential memory) so that the description is true regardless of the technology used to implement the system…an essential model.*
> *It is also possible to describe a system as actually realised by a particular technology…an implementation model.*
> *The implementation model is defined as an elaboration of the essential model that contains enough detail to permit a successful implementation with a particular technology." [41]*

We extend these concepts to the domain of systems architecture by introducing the *essential architecture* and the *implementation architecture*. In addition to these two architectures we also introduce the concept of many intermediate architectures. In all cases, these architectures are actually represented using the UTS [26] terms summarised in section 2. Each of the essential, intermediate and implementation architectures represent differing levels of abstraction in modelling the system.

4.3.1 Essential Architecture

The *essential architecture* is the most abstract representation that a particular project will use of the architecture. It utilises abstraction to help highlight key system properties, architectural components and component interaction. The essential A-Model(s) are the primary model(s) about which one can reason about to ensure that the system is capable of meeting its requirements. Further essential modelling concepts may be drawn from systems engineering and systems theory literature to help establish the essential A-Model [3, 29, 41]. Our approach is used to formalise the concepts associated with the essential A-Model and then provides a basis on which to reason about the feasibility of the proposed architecture.

The most important aspect of an essential architecture is that, by definition, it contains no implementation details. As Ward and Mellor state, the system as described by an essential architecture could equally be implemented by humans manually executing the required processes as it could by a computer system [41]. The essential architecture should give no indication of what technology should be used in the final implementation.

4.3.2 Intermediate Architecture

Depending on the complexity of the system being modelled, there may be multiple *intermediate architectures*. The intermediate architectures describe the system in successively more detail than the (essential) requirements driven essential architecture. The intermediate A-Models are a primary tool in refining the system down to a subsequent implementation architecture that is capable of meeting the system's functionality, performance and quality (including evolvability) requirements. Our approach is used to develop the intermediate A-Models and provides a basis on which to continue to reason about the feasibility of the proposed architecture, and its relation to the essential architecture. In relation to the essential architecture, we are especially interested in the intermediate architecture being a correct refinement, which shows promise in meeting the non-functional requirements.

One important point to raise here is that by having intermediate architectures one can approach the A-Design of a system at whatever level of abstraction the designers are comfortable with, or have data with which to populate the A-Model [3, 29, 41]. The merits of this flexibility concept in our A-Design approach will be further discussed in section 5.

4.3.3 Implementation Architecture

A system's essential and intermediate A-Models are important abstractions, however they do not consider the system solution sufficiently with respect to implementation issues – it is the final *implementation architecture* that delivers the final, specified functionality and capability [39, 43]. Our approach is used to ensure that the architectural solution chosen is appropriate and feasible given the skills and technology available.

The final implementation architecture is not the end of the detailed design process, it is in fact, the beginning. The implementation architecture components, connections and constraints are now ready to be implemented as (e.g.) collections of classes.

4.4 A representation of A-Design

Given the definitions for architecture-based design (A-Design) and the essential, intermediate(s) and implementation A-Models as discussed above, an approach for A-Design can now be presented that facilitates the mapping between each of the A-Models.

Figure 3 illustrates the important concepts in our approach for performing A-Design. It can be seen that A-Design can occur in two "dimensions": *horizontal* and *vertical*, and that each of these dimensions is bi-directional. Each level of abstraction (essential, intermediate(s) or implementation) satisfies a certain subset of the system's requirements. The requirements shall either be fulfilled directly at that level of abstraction, or indirectly via the fact that the architecture at the current level of abstraction is a faithful interpretation of those architectures at a higher level of abstraction. Thus, at the implementation architecture, all requirements shall be fulfilled [31].

It is important to note that while each vertical refinement of the A-models in Figure 3 is shown as a graphically similar A-model this is not necessarily the case. As represented graphically and described in [12], for successive A-models "a given element from one space can map to zero, one, or more elements in the lower level space".

5. A Practical A-Design Approach

This section presents a practical approach to architecture-based design (A-Design). Firstly, the requirements for A-Design (the necessary capabilities of a A-Design method) are presented. The details of a practical A-Design method follow. The section finishes by discussing how this practical A-Design method satisfies the requirements of A-Design.

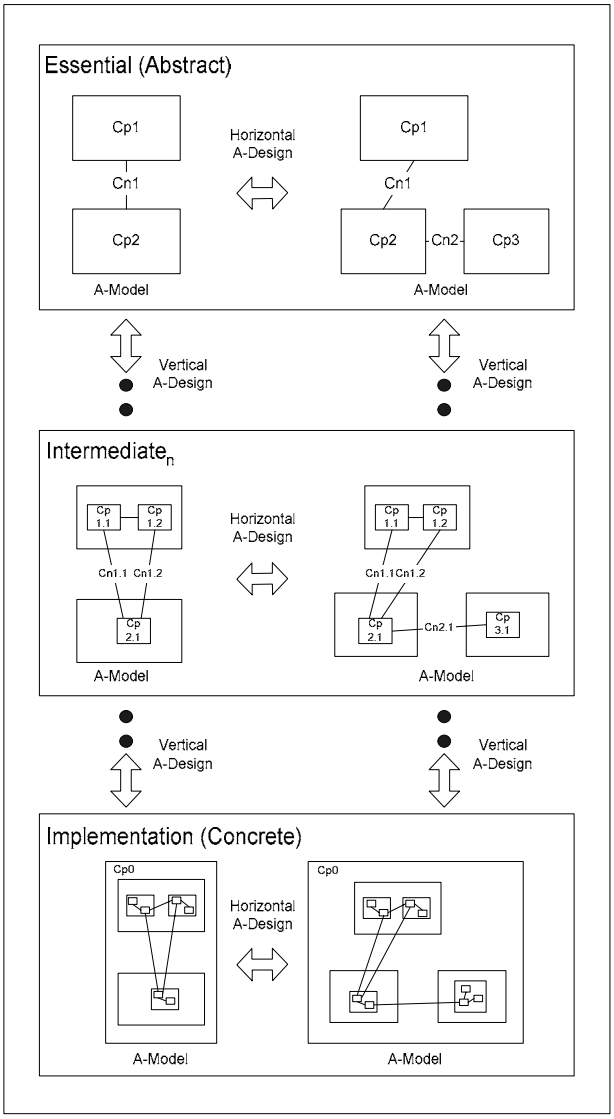

Figure 3 – Architecture-based design (A-Design) using architectural models (A-Models) of different levels of abstraction

5.1 Requirements of A-Design

Following from the previous discussion, and definitions, we propose that the following requirements need to be met in order to produce a practical A-Design process that will aid the designer in both the development and the evolution of Computer Based Systems. The A-Design approach must:

1) Be scalable and practical for large, heterogeneous systems.
2) Provide the ability to begin design with an A-Model at any level of abstraction.
3) Provide the ability to design in any dimension and direction (see section 4).
4) Support "long term design", that is, evolution
5) Ensure that both functional and *non-functional* requirements are met.

6) Be rigorous, yet flexible, with no specified ordering of horizontal or vertical A-Design steps.

5.2 Approach

This section describes in steps a process for practical A-Design. The input to the A-Design process is an A-Model, and the output is simply another A-Model that has been refined or abstracted depending on the dimension in which refinement is occurring.

There are two "pre-steps", or essentially tasks that are assumed to have been done prior to use of the practical A-Design approach, as follows:

Pre-Step A (A-Model Population / A-Discovery):
This pre-step involves gathering the appropriate "information" from the most appropriate available sources. Examples include requirements documents, especially constraints and specified equipment, source code, design documents and interviews with system architects / designers. Once this information is parsed (either manually or automatically) a collection of elements (but not necessarily A-Elements) will exist. A-Discovery is concerned with reasoning about the gathered elements and deciding which of those are legitimate A-Elements and which are not. Thus, after gathering the elements they must be filtered so as to keep only the A-Elements. Once this is done we have the initial architecture model, designated A-Model$_i$. A-Model$_i$ may represent the system at any level of abstraction (from Essential to Implementation).

Pre-Step B (Requirements):
A requirements analysis has to have been completed before the A-Design can commence. The requirements need not be completely defined from the outset, however when the A-Design takes place it will simply work off whichever requirements are defined at that stage. As such the approach could be used in many different system development life cycles, from traditional waterfall to evolutionary, all of which have a different notion as to what stage and level of completion the requirements analysis shall be completed before embarking on the subsequent life cycle stages.

Once these pre-steps have been completed, the main steps of the architecture-based design approach can commence. The input to the approach is a certain A-Model, A-Model$_i$. The output from an iteration of the approach is the next A-Model, A-Model$_{i+1}$. The composition of the new A-Model depends on which direction and dimension A-Design is taking place. For example if one is abstracting (vertical A-Design moving upwards) the A-Model might typically contain fewer A-Elements and be generally simpler, whereas if one is refining (vertical A-Design moving downwards) the A-Model might typically contain more A-Elements and be generally more complex.

The main steps of the approach are shown diagrammatically in Figure 4 and are as follows:

Step 1 (Initial Evaluate and Prove):
This step is important as it sets the baseline "status" for A-Model$_i$. This step gives the approach the ability to know where one is coming *from*, in order to guide where one is going *to*. For example, assume at a high level of abstraction, an architecture exhibited a strong peer-to-peer architectural shape, however an implementation requirement is the specification of a particular client-server (shaped) database. The evaluation of the architecture when the client-server implementation detail is added would be flagged as a "mismatch".

Essentially, the aim is to prove whether the functional requirements (R_F) are met, and to evaluate whether the non-functional requirements (R_{NF}) are met (for more details see step 4). Regardless of whether the requirements are met or not, this evaluation, or performance index (PI), is fed to the A-Principles "knowledge base". This knowledge base is used for the generation of new A-Models, that is, it is used to gauge how successful previous modifications to the A-Models have been, and to then guide the proposal of new A-Models.

Step 2 (Propose New A-Model):
This step involves generating a proposed alternative to the current A-Model$_i$, designated A-Model$_{i+1}$. The generation of the next A-Model$_{i+1}$ is guided by the A-Principles knowledge base (as discussed above). Given A-Model$_i$, the PI guides the designer in making the appropriate aggregations, substitutions and decompositions depending on the dimension and direction being refined (see section 4).

Step 3 (Evaluate and Prove):
This step is a repetition of the activities of step 1 and involves checking A-Model$_{i+1}$ against the requirements, both functional and non-functional. It must be proven that the functional requirements are met. It is at this point in the approach that existing methods of architectural refinement could be used. The specific method used would vary depending on how the functional requirements had been expressed [5, 31]. Typical non-functional requirements evaluated in this step are performance, evolvability and openness.

Once these proofs and evaluations are complete, we can see whether A-Model$_{i+1}$ satisfies the requirements. The results of the evaluation are fed back into the A-Principles PI so as to enhance the "knowledge" contained. Should the evaluation prove that the requirements have been met, the fact that the changes from A-Model$_i$ to A-Model$_{i+1}$ resulted in a successful refinement are incorporated into the knowledge base. The same is true for the reverse case - the changes did not result in the requirements being met, and this is also fed back into the knowledge base. If the answer is "No" then the process is repeated from step 2 again until we are successful in meeting the requirements and we have the refined A-Model, designated A-Model$_{i+1}$.

It is evident, via step 1 of the approach, that we have the ability to enter the A-Design process at whichever level of abstraction is most appropriate. Obtaining the initial A-Model$_i$ from source code will result in a refinement process that begins at a much more concrete level of abstraction than one that begins with a high level architectural description.

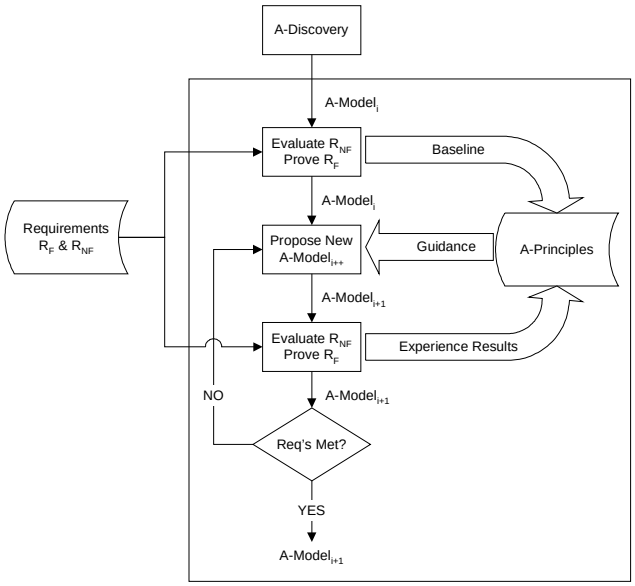

Figure 4 – An Iteration of the Architecture-based design (A-Design) Approach

Within this approach, A-Design can proceed in any dimension and any direction (horizontal moving left and right, and vertical moving up and down respectively). This is partly due to the evaluation method described in step 1 and 3 which allows any new model, A-Model$_{i+1}$ to be evaluated against the functional and in particular the non-functional requirements of the system.

The benefit of this to the designer is that it gives them the ability whilst typically developing a new system, to design vertically and down in order to approach a more concrete model. In addition, for an as-built system which is required to evolve in a certain way, it gives the designer the ability to enter the A-Design process with a typically more concrete model, and to abstract vertically and upwards until the model is sufficiently abstract to reason about and make the required changes. Once this reasoning is complete and satisfactory, the A-Design process can proceed in the reverse direction (that is, vertically and down) until a suitably concrete model is again obtained. This "round trip" of abstracting up, reasoning, making alterations and then refining down is an approach that is very well suited to the long term evolution of a system.

Consequently, the approach is very flexible in allowing the designer the freedom to enter at any level of abstraction, and to move in any dimension and direction as required by the designing task at hand. However, at the same time, the approach is rigorous in that at each A-Design step the new model is evaluated against the overall requirements of the system to ensure that no unsuitable or unwanted changes have been made.

Non-functional requirements and their evaluation are fundamental to this approach and it is in this area that the practical A-Design approach described here differs the most from existing refinement methods [5, 18, 31].

The approach is tied heavily to the satisfaction of requirements, functional and importantly non-functional. Consequently, it gives the designer confidence that they are not only producing an architecture that is a valid refinement or abstraction of their starting point, but additionally that it is a "good" architectural solution.

6. Future Work and Conclusion

There are three aspects to the future work. Firstly, to incorporate related work such as co-design [35], and other methods such as MASCOT. Secondly, to include into the approach the idea of evolution as a third dimension of A-Design. We also need to develop a formal method for evolution. Finally, we need to incorporate the whole approach into the ABACUS tool suite [1].

In conclusion, we have developed a practical approach to the architecture-based design (referred to as A-design) which aims to simultaneously satisfy the functional and non-functional properties of a system. This approach is based upon the various architectural refinement calculi.

7. Acknowledgements

This work was undertaken as part of an Avolution Pty Ltd sponsored project and funding was also furnished in part by the Australian Research Council (ARC) in the form of an Australian Postgraduate Award (APA) scholarship for Mark Denford.

8. References

[1] ABACUS, http://www.avolution.com.au.

[2] Abrial, J-R., *the B-book – assigning programs to meanings*, Cambridge University Press, 1996.

[3] Alexander C., *Notes on the Synthesis of Form*, Harvard University Press, Cambridge MA, 1964.

[4] Allen, R., J., *A Formal Approach to Software Architecture*, PhD Thesis, Carnegie Mellon University, CMU Technical Report CMU-CS-97-144, May 1997.

[5] Bolusset, T., Oquendo, F., *Formal Refinement of Software Architectures Based on Rewriting Logic*, International Workshop on Refinement of Critical Systems: Methods, Tools and Experience, Grenoble, France, 2002.

[6] Büchi, M., Sekerinski, E., *Formal Methods for Component Software: The Refinement Calculus Perspective*, Second Workshop on Component-Oriented Programming, Jyväskylä, June 1997.

[7] Carrington, D., Duke, D., Duke, R., King, P., Rose, G., Smith, G., *Object-Z: An Object-Oriented Extension to Z*. FORTE 1989: 281-296.

[8] Chaudet, C., Oquendo, F., *π-Space: A Formal Architecture Description Language Based on Process Algebra for Evolving Software Systems*, Proceedings the Fifteenth International Conference .on Automated Software Engineering (ASE'00), Grenoble, France, September 2000.

[9] Denford M., O'Neill T., Leaney J., *Architecture-based Visualisation of Computer Based Systems*, Proceedings of ECBS'02, Lund, Sweden.

[10] Dijkstra, E. W., *Why Correctness Must Be a Mathematical Concern*, in Boyer, R. S. and J. S. Moore (eds.), The Correctness Problem in Computer Science, Academic Press, 1981.

[11] Egyed, A., Grönbacher, P., Medvidovic, N., *Refinement and Evolution Issues in Bridging Requirements and Architecture – The CBSP Approach*, First International Workshop From Software Requirements to Architectures (STRAW'01), Toronto, Canada, May, 2001.

[12] Egyed, A., Medvidovic, N., *Consistent Architectural Refinement and Evolution using the Unified Modeling Language*, First Workshop on Describing Software Architecture with UML, Toronto, Canada, May, 2001.

[13] Garlan, D., *Style-based Refinement for Software Architectures*, Proceedings of the Second International Software Architecture Workshop (ISAW2), San Francisco, October 1996.

[14] Guo, Y.G., Atlee, J.M., Kazman, R., *A Software Architecture Reconstruction Method*, Proceedings of First Working IFIP Conference on Software Architecture (WICSA1), San Antonio, TX, USA, February, 1999.

[15] Hayes, I.(ed.), *Specification Case Studies*, Prentice-Hall, 1987.

[16] Heales J., *Evolutionary and Revolutionary Maintenance of information Systems: A Theoretical and Empirical analysis*, PhD Dissertation, Dept. of Commerce, The University of Queensland, Australia, 1998.

[17] Hoare, C. A. R., The Axiomatic Basis of Computer Programming, Communications of the ACM, 12 (10), October, 1969

[18] Hoare, C.A.R., *Communicating Sequential Processes*, Prentice Hall, 1985.

[19] Horowitz, B.M., *The Importance of Architecture in DOD Software*, The MITRE Corporation, Report M91-35, July 1991.

[20] IBM Pty Ltd, *IBM Insurance Application Architecture*, http://www-1.ibm.com/industries/financialservices/solution/SOLUTIONS_30269.html, 2002.

[21] *IEEE Standard 1233: IEEE Guide for Developing System Requirements Specifications*, IEEE Computer Society, 1998.

[22] *IEEE Standard 1471: IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*, IEEE Computer Society, 2000.

[23] Jones, C. B., *Software Development - A Rigorous Approach*, Prentice-Hall, 1979.

[24] Kazman, R., O'Brien, L., Verhoef, C., *Architecture Reconstruction Guidelines*, CMU/SEI-2001-TR-026, Technical Report, Software Engineering Institute, Carnegie Mellon University, August 2001.

[25] Larch, http://nms.lcs.mit.edu/Larch/

[26] Leaney, J. et al., *Current Developments in the Definition and Description of System Architectures - A University of Technology, Sydney (UTS) Architecture-based Engineering Research Program Architecture Working Group (AWG) Position Paper*, Internal UTS report 2002-R-01.

[27] Leaney, J. et al, *Measuring the Effectiveness of Computer Based Systems: an Open System Measurement example*, Proceedings of ECBS'01, pp 179-189.

[28] Lehman M.M. & Belady L.A., "A model of large program development", *IBM Systems Journal*, vol. 15, no. 3, 1976, pp 225-251.

[29] McMenamin, S.M. & Palmer J.F., *Essential systems analysis*, Yourdon Press, Englewood Cliffs (N.J.), 1984.

[30] Medvidovic N., Taylor R., *A Classification and Comparison Framework for Software Architecture Description Languages*, IEEE Transactions on Software Engineering, Volume 26, Number 1, 2000, pp. 70-93.

[31] Moriconi, M., Qian, X., Riemenschneider, R. A., *Correct Architecture Refinement*, IEEE Transactions on Software Engineering, Volume 21, Number 4, 1995, pp. 356-372.

[32] O'Neill, T., Leaney, J., Martin, P., *Architecture-Based Performance Analysis of the COLLINS Class Submarine Open System Extension (COSE) Concept Demonstrator (CD)*, Proceedings 7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems, Edinburgh Scotland, 2000, pp. 26-35.

[33] Perry D.E., "Dimensions of Software Evolution", in *Proceedings of 1994 Conference on Software Maintenance*, IEEE, 1994, pp. 296-302.

[34] Rechtin, E., "Systems Architecting : creating and building complex systems", Prentice Hall, 1991

[35] Schulz, S., Rozenblit, J.W., Mrva, M. and K. Buchenrieder, Model-Based Codesign, IEEE Computer, 13(8), 1998.

[36] Rowe D., Leaney J., Lowe D., *Defining systems evolvability – a taxonomy of change*, Proceedings of ECBS'98, Jerusalem, Israel.

[37] Rowe et al., "IEEE ECBS'99 TC Architecture Working Group (AWG) Discussion Paper" Proceedings of ECBS'99, pp. 359-364.

[38] Simpson, H.R., "Architecture for Computer Based Systems", IEEE Publication 0-8186-5715-4/94, 1994.

[39] Thome B. (ed), "Systems engineering: principles and practice of computer-based systems engineering", John-Wiley and Sons, Chichester (N.Y.), 1993.

[40] Unified Modeling Language, Rational Software Corporation, most recent updates of UML are available via WWW http://www.rational.com

[41] Ward, P.T., Mellor, S.J., *Structured Development for Real-Time Systems: Volume I*, Yourdon Press, Englewood Cliffs (N.J.), 1985.

[42] Wordsworth, J. B., The CICS application programming interface definition, Z User Workshop, Oxford 1990

[43] Wymore A.W., *Systems Engineering Methodology for Interdisciplinary Teams*, John Wiley and Sons, New York, 1976.