Non-Functional Refinement of Computer Based Systems Architecture

Mark Denford, John Leaney, Tim O’Neill
Institute for Information and Communication Technologies
University of Technology, Sydney

Abstract

Architecture based refinement is an important technique for ensuring efficiency, effectiveness and correctness in the practical design of complex computer based systems. With very few exceptions, current methods of architectural refinement focus on functional behaviour and fail to address non-functional requirements throughout the refinement process. A best practices approach to refinement would address both functional and non-functional requirements such that the refinement of an abstract into a concrete (implementation) architecture ensures that both sets of requirements are met.

We propose a method that focuses on the non-functional requirements while still addressing the functional requirements throughout refinement. The method has a formal underpinning in abstract data types (based on term rewriting) which are used to represent the architectures throughout the refinement process and to place pre and post conditions on the refinements. In addition to this, the method uses non-functional requirement calculators to check the non-functional qualities of the architecture as refinement proceeds.

Reflection on the practice of the method suggests that it may be possible to extend the architectural style idea to provide reusable refinement schema for the design of certain non-functional qualities into architectural patterns. The example considers reliability and performance in the refinement of a client server architectural pattern.

The method does not aim to replace or fully automate the work of the designer. It aims to augment the design process and aid the designer in performing their tasks. It seeks to provide certain guidance for the designer that will help them make the right design decisions, and correct certain classes of errors.

1. Introduction

The design of large computer based systems has always been one of the most challenging tasks to befall the system engineer. Rechtin, in his book of reflections and guidance for system/software architects [25], highlights that large system design is always a task dominated by complexity, requiring excellent abstraction skills and experience. Over the last 14 years, in the academic and research community, much gusto has been put into methods, techniques and algebra for systems/software architecture.

2. Inspiration and ‘revenir du temps perdu’1

This section presents a summary of current related work in the area of architectural refinement of computer based systems. We argue along with Barber et al [5] that effective refinement has to be technology and domain specific, and that non-functional and functional requirements must be refined in concert. Further there seem to be some important lessons to be learnt by early attempts to formally describe systems at an abstract level.

The focus of architecture based refinement is clearly on the refinement of the structure of the system, and the satisfaction of non-functional requirements [15].

We would like to return to one of the first architecture description languages (ADLs), designed for describing the organization, or structures, of computers. PMS [7, 27]. In the opinion of the second author, PMS provided the perfect abstract representation for computer design.2 Why was it ‘perfect’? It had features which modelled the few, essential components of a computer: the Processor (P), the Memory (M) the Switch (S) (the switch represents one of the many options for connecting the other components, mostly the P and M). There are three other symbols: K (the system controller), L (the link, a specialised connection), T (a transducer to transform information, the transducer could be a serial to parallel converter, an analogue to digital converter, etc) Subscripts are used to denote detailed function. For example, Mp denotes primary memory (commonly RAM) Ms denotes secondary memory (‘hard’ disc, CD, etc). PMS has four components (P, M, K, T) and two connections (S, L). This small number of types of components and connections certainly accords with

1 (remembrance of times lost) The history of software engineering, and to a lesser extent CBSs, is a sorry tale of the latest and the best, with only very occasional acknowledgement of history, which we are bound to repeat – often poorly. The half-life of reused knowledge appears, based on citations, etc. to be about 5 to 8 years. In a discipline with about 60 years of experience, this is surely youthful arrogance.

2 PMS was used throughout the late 1980s and early 90s as an ADL by the second author to model combat systems, stock market distribution, etc.
Occam’s razor\(^3\) when one considers the number of computers which have been built in the world, and which can be understood in PMS by any graduate computer engineer. Constraints are represented by specific annotations to both components and connections. The description of any computer is a matter of choosing from a very small set of well defined components followed by topology, and the choice of either the S or L connection. The purpose of the components (e.g., memory type) is then added as subscript. It then remains to start to nominate constraints which will define all encompassing properties such as word length, memory size, etc. PMS has been formally extended in [4] to enable proof of a refinement of a PMS structure into a more concrete ISP/RTL representations.

PMS clearly exhibits the systems engineering principles of domain restriction; namely restricting domains of technology and domains of application. For PMS, the application domain is the computer, the technology is digital logic. Domain and technology restriction give clear guidance to the designer, but without restricting options.

While being historical, it seems important to reflect on the very first general architectural representation and ADL, namely MASCOT [1, 24, 28]. MASCOT is still in use and in development ([2, 29]). Again, the success of MASCOT was in its few types of connections (but no restriction on components), its focus on interconnection (Philips' networks for real time programming). Its refinement procedures consist of a small set of abstract connections leading to a finite, but fairly large set of concrete connections [29], and, a specified technology domain (real-time systems) and implicit application domain (aerospace systems). Domain and technology restriction are important to making a system useable in practice [18].

A third example of architecture representation is that of Hatley and Pirhbai [17]. They provide a representation, and a refinement method, known as RTSS. Their work is important in linking requirements and architecture, and has been used in industry over the ensuing years.

The question may well be asked, why are we publishing anything other than further work on PMS, MASCOT and RTSS? Suffice it to say, we are learning from these systems, and, may in the future “…arrive where we started and know the place for the first time” [12].

Learning from these three successful systems, and, extracting a few of their notable features, below, as a guide, gives us a framework within which to examine the advances in representation and refinement in the last 10 years.

(Some of) the features of MASCOT, PMS and RTSS which made them ‘use-ful’ are:

- The few types of components and connections – this leads to fewer opportunities for misinterpretation.
- The expressive power – the great computing machines of the world and many, many realtime systems have been described simply and elegantly in PMS, MASCOT and RTSS.
- The ease with which both formal, and rigorous, refinement may proceed.
- The accessibility to the appropriate professional (the graduate computer engineer/ the practising system architect).
- The use of domain and technology restriction.
- Explicit refinement of non-functional requirements (NFRs). Time Performance is the dominant NFR refined in these systems.

Before using the framework to examine aspects of refinement, a classification of the mathematical bases for current refinement and representation techniques seems relevant.

### 2.1 Mathematical bases of Refinement

Neno Medvidovic [20] has provided an extensive review paper in the field of ADLs. However, in his review, only Moriconi and SADL had a focus on representation and refinement. Since then there have been many separate schemes on refinement (at least 18) (at least 11 in the review included in [10] plus PMS, MASCOT, Barber [5], Ergodmus [13], Payne [22], Philips [23], Christoph [9], which represents another seven discussed herein). The work reviewed seems to fit into the following general classifications: graph rewriting, first order predicate logic, (theories, Z, (homomorphisms and isomorphisms)), process algebra (CSP, \(\pi\)-calculus) and term rewriting (Peano, Larch).

Graph rewriting techniques have clearly a major role. They are apparently scalable and satisfy ease of use for the practising system architect. They have been used successfully in transformation work [14, 30]. However, their semantics can be a delusion as they refine on allowable connections and types alone [9, 14]. Without any supporting further semantics, graph rewriting alone allows one to specify practically any sort of rewrite, regardless of what it might actually mean to the architecture. Further semantics are needed to help specify this meaning.

Inevitably first order predicate logic sprang forth [3] with its origin in program modelling. Unfortunately, it has brought with it the difficulty of general adoption by the practice community, beyond preconditions and postconditions.

Process algebra [3] has been used to model interaction, especially connections. It also suffers from the lack of uptake based on its previous life in program modelling.

---

\(^3\) One should not increase, beyond what is necessary, the number of entities required to explain anything.
2.2 Review against the framework

Following is a review of some current research systems against the framework extracted from current successful industrial systems.

2.2.1 Domain and technology restriction

Barber [5, 6] both support domain restriction, and their refinement method is strongly based on the domain architecture.

The work on patterns, including pipe and filter refinement [21, 23] exhibits technology restriction.

2.2.2 Refinement of non-functional requirements

Barber et al [5] describe a framework for iterative refinement with model based checking of NFRs at each stage. Their refinement is strongly driven by the NFRs. They also provide heuristics to guide the NFR refinement.

Rosa et al [26] in their system PARMENIDES provide a template for NFRs, and refinement templates for assignment to components. Their refinement is strongly driven by the functional requirements.

2.2.3 Accessibility to the professional and expressive power

Barber et al [6] would seem to have the strongest claim in this regard. However, there does not appear to be many field trials, if any, beyond [6].

Expressive power is again not checked as there is so little evidence of use in practice. In fact, there appears to be no complete example refined according to a practical specification in the literature.

2.2.4 Few types of components/connections

Again, Barber [5], via domain restriction would seem to be the strongest in this regard.

2.2.5 Further exploration

We find that there is a further exploration worth entering into on the matters of:

- explicitly refining a variety of non-functional requirements, using term rewriting augmented by pre/post conditions;
- refining the functional requirements in concert with the non-functional requirements;
- a practical, formal and rigorous refinement method which works efficiently, effectively and correctly;
- application and technology domain restriction;
- considerable field work and validation of the new ideas of the last 14 years.

3. A Schema for Refinement Focussing on Non-Functional Requirements

Throughout the worked example (presented later) refinements are made to a given architectural model. These refinements can be specified in terms of the pre and post conditions (on some mathematical formalism of the architecture, in our case Abstract Data Types) on the “before” and “after” architectural models respectively.

However in addition to this, while refining, it is imperative to know the meaning and effect of making certain refinements. Our refinement schema attempts to address this. Firstly, the refinements are valid within a certain application and technical domain, that is they are domain-specific [5]. Secondly, by using the refinement in a certain domain, there are certain non-functional qualities that we can reason about as resulting from the application of the refinement. Finally, there is a concept of certain obligations that may arise from using these refinements.

The domain of the refinements is the type or types of systems within which the refinements have meaning. A given refinement may be valid across multiple domains and as research into this area continues general refinements may emerge that can be used across all domains. However, for the refinements illustrated below, the domain is explicitly specified.

Given that non-functional requirements are dependent on the structure of the system [15] by applying structural changes in certain (specified) areas of an architecture we can reason about the effect of this on the non-functional qualities of the system.

The obligations that we speak of are currently specified in natural language, and describe the obligations that the design must fulfill to fully incorporate the refinement.

To summarise, a given refinement step will be specified in terms of the ADT pre and post conditions on the architecture, the domain in which it is valid, the effects on the non-functional qualities of the system, and the resulting obligations on the design. We can represent this as a schema as follows:

Refinement(ADT_Preconditions, ADT_Postconditions, Domain, NF-effect, Obligations).

Without such a schema, if we simply specified the refinements in terms of the architectural representation before and after the refinement takes place we have no concept of the meaning of making such a refinement. Such an approach would be analogous to a simple graph-rewriting approach only concerned with pattern-matching to make suitable pattern substitutions. That is, it would be context-free and meaningless in terms of the overall qualities of the architecture.
The following sections first present the Abstract Data Types that we use within the schema and then using these ADTs we present an example of a full schema.

3.1 An Abstract Data Type Formalism of an Architecture

As a means of formalising the representation of the architectures with which we work, we have developed Abstract Data Types (ADT). These ADTs describe the following entities: Architectures, Components, Connections, Ports, and Types.

The ADTs are integral to our refinement method as they are used to represent the architectures throughout the refinement process, and to specify pre and post conditions on certain types of refinement steps. That is, the ADTs are used as a “guard” before a refinement step can take place, and as a “guarantee” that certain (structural) conditions will be met afterwards. This combination of ADTs with pre and post conditions is similar to the way Larch [16] operates.

3.1.1 ADTs

We have represented Architectures, Components, Connections, Ports and Types in Abstract Data Types. The ADTs contain functions and axioms for each architectural element. For the sake of brevity we do not present functions and axioms that are not critical to the general understanding or to the proofs and reasoning presented in this paper. These ADTs are presented and discussed in this section.

**Table 1 - ADT for an Architecture**

<table>
<thead>
<tr>
<th>ARCHITECTURE (arch for short)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE arch</td>
</tr>
<tr>
<td>imports comp</td>
</tr>
<tr>
<td>conn</td>
</tr>
<tr>
<td>Boolean</td>
</tr>
<tr>
<td>String</td>
</tr>
<tr>
<td>FUNCTIONS</td>
</tr>
<tr>
<td>New(string)-&gt; arch</td>
</tr>
<tr>
<td>Name(arch) -&gt; string</td>
</tr>
<tr>
<td>AddComp(arch, comp) -&gt; arch</td>
</tr>
<tr>
<td>AddConn(arch, conn) -&gt; Boolean</td>
</tr>
<tr>
<td>HasComps(arch) -&gt; Boolean</td>
</tr>
<tr>
<td>HasConn(arch) -&gt; Boolean</td>
</tr>
<tr>
<td>AXIOMS</td>
</tr>
<tr>
<td>for all a, a':arch ; cp.comp ; cn.conn ; s:string, let</td>
</tr>
<tr>
<td>1. Name(New(s)) = s</td>
</tr>
<tr>
<td>2. HasComps(New(s)) = false</td>
</tr>
<tr>
<td>3. HasComps(AddComp(a, cp)) = true</td>
</tr>
<tr>
<td>4. HasConn(Net(s)) = false</td>
</tr>
<tr>
<td>5. HasConn(AddConn(a, cn)) = true</td>
</tr>
</tbody>
</table>

**Table 2 - ADT for a Component**

<table>
<thead>
<tr>
<th>COMPONENT (comp for short)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE comp</td>
</tr>
<tr>
<td>imports cpPort</td>
</tr>
<tr>
<td>cpType</td>
</tr>
<tr>
<td>Boolean</td>
</tr>
<tr>
<td>String</td>
</tr>
<tr>
<td>FUNCTIONS</td>
</tr>
<tr>
<td>New(string, cpType)-&gt; comp</td>
</tr>
<tr>
<td>Name(comp) -&gt; string</td>
</tr>
<tr>
<td>Type(comp) -&gt; cpType</td>
</tr>
<tr>
<td>AddPort(comp, cpPort) -&gt; comp</td>
</tr>
<tr>
<td>HasPorts(comp) -&gt; Boolean</td>
</tr>
<tr>
<td>AXIOMS</td>
</tr>
<tr>
<td>for all cp, cp', cp'':comp ; ccpp:cpPort ; s:string ; t:cpType, let</td>
</tr>
<tr>
<td>1. Name(New(s, t)) = s</td>
</tr>
<tr>
<td>2. Type(New(s, t)) = t</td>
</tr>
<tr>
<td>3. HasPorts(New(s)) = false</td>
</tr>
<tr>
<td>4. HasPorts(AddPort(cp, cpp)) = true</td>
</tr>
<tr>
<td>5. HasConn(AddComp(cp, cp', cpp') = true</td>
</tr>
<tr>
<td>6. ContainsComp(New(s, t), cp) = false</td>
</tr>
<tr>
<td>7. ContainsComp(AddComp(a, cp)) = true</td>
</tr>
<tr>
<td>8. ContainsConn(New(s, cn)) = false</td>
</tr>
<tr>
<td>9. ContainsConn(AddConn(a, cn)) = true</td>
</tr>
</tbody>
</table>

**Table 3 - ADT for a Component Port**

<table>
<thead>
<tr>
<th>COMPONENT PORT (cpPort for short)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE cpPort</td>
</tr>
<tr>
<td>imports cnPort</td>
</tr>
<tr>
<td>cnPort</td>
</tr>
<tr>
<td>Boolean</td>
</tr>
<tr>
<td>String</td>
</tr>
<tr>
<td>FUNCTIONS</td>
</tr>
<tr>
<td>New(string) -&gt; cpPort</td>
</tr>
<tr>
<td>Name(cpPort) -&gt; string</td>
</tr>
<tr>
<td>Attach(cpPort, cnPort) -&gt; cpPort</td>
</tr>
<tr>
<td>HasAttachments(cpPort) -&gt; Boolean</td>
</tr>
<tr>
<td>AXIOMS</td>
</tr>
<tr>
<td>for all cpp, cp':cpPort ; cn:cnPort ; s:string, let</td>
</tr>
<tr>
<td>1. Name(New(s)) = s</td>
</tr>
<tr>
<td>2. HasAttachments(New(s)) = false</td>
</tr>
<tr>
<td>3. HasAttachments(Attach(cpp, cpp)) = true</td>
</tr>
<tr>
<td>4. IsAttachedTo(New(s), cpp) = false</td>
</tr>
<tr>
<td>5. IsAttachedTo(Attach(cpp, cpp), cpp) = true</td>
</tr>
</tbody>
</table>
### Table 4 - ADT for a Connection

<table>
<thead>
<tr>
<th>CONNECTION (conn for short)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TYPE</strong> conn</td>
</tr>
<tr>
<td>imports</td>
</tr>
<tr>
<td>cnPort</td>
</tr>
<tr>
<td>cnType</td>
</tr>
<tr>
<td>Boolean</td>
</tr>
<tr>
<td>String</td>
</tr>
</tbody>
</table>

**FUNCTIONS**
- New(string) -> conn
- Name(conn) -> string
- Type(conn) -> cnType
- AddPort(conn, cnPort) -> conn
- HasPorts(conn) -> Boolean
- ContainsPort(conn, cnPort)

**AXIOMS**
- for all cn, cn':conn ; cn:cnPort ; s:string ; t:cnType, let
  1. Name(New(s)) = s
  2. Type(New(s, t)) = t
  3. HasPorts(New(s)) = false
  4. HasPorts(AddPort(cn, cnp)) = true
  5. ContainsPort(New(s), cnp) = false
  6. ContainsPort(AddPort(cn, cnp), cnp) = true

The ADTs for the Connection Ports are excluded as they are essentially the same as those for the Component Ports.

It is worth making explicit a few points regarding the architectural model, as described in the ADT’s. It can be seen in Table 1 above that an architecture contains both components and connections. Table 2 shows that a component contains ports and other components (i.e. sub-components). Table 3 shows that component ports are attached to connections ports, and it is via this sub-mechanism that components are related to other components via connections. In addition to these structural characteristics, we note also from Table 2 that in our architectural model components and connections are always of a certain type. That is, when a component is created it must be given a certain type. These concepts are reasonably well accepted as common in most architecture description languages (ADLs) [20].

We have built a tool that provides the above functions and enforces the above axioms when constructing architectures. As a result, this tool provides the first level of checking throughout the refinement process as new architectures are proposed, we can be sure that they conform to our architectural model, and ADTs in particular.

### 3.2 An Example Refinement Schema

This refinement schema aids reliability of the system. The domain is that of the worked example, specifically a distributed, high performance, reliable file server system.

Speaking more generally, however, the domain could be described as “distributed client / server systems”.

In these sorts of client / server systems reliability in the servers is a common non-functional requirement and “replicating” the servers is a common way to address this. The strict evaluation of this reliability is discussed later under the details of the framework and illustrated in the worked example.

![Figure 1 – Elements and Topology of Initial Architecture and Final Architecture](image)

We can specify both the initial and final architectures in terms of the ADT functions used to create each architecture.

**ADT Representation of initial architecture:**

\[
\begin{align*}
A_1 := & \quad \text{arch.New("initial arch")(\text{A1})} \\
CP1 := & \quad \text{comp.New("comp1", Server)(\text{CP1})} \\
CP2 := & \quad \text{comp.New("comp2", Client)(\text{CP2})} \\
CPP1 := & \quad \text{cpPort.New("cpport1")(\text{CPP1})} \\
CPP2 := & \quad \text{cpPort.New("cpport2")(\text{CPP2})} \\
CN1 := & \quad \text{conn.New("conn1", ClientServerCall)(\text{CN1})} \\
CNP1 := & \quad \text{cnPort.New("cnport1")(\text{CNP1})} \\
CNP2 := & \quad \text{cnPort.New("cnport2")(\text{CNP2})} \\
\end{align*}
\]

AddComp(A1, CP1)
AddComp(A1, CP2)
AddPort(CP1, CPP1)
AddPort(CP2, CPP2)
AddConn(A1, CN1)
AddPort(CN1, CNP1)
AddPort(CN1, CNP2)
Attach(CPP1, CNP1)
Attach(CPP2, CNP2)

**ADT Representation of final architecture:**

\[
\begin{align*}
A_2 := & \quad \text{Arch.New("final arch")(\text{A2})} \\
CP1\_1 := & \quad \text{comp.New("comp1\_1", Server)(\text{CP1\_1})} \\
CP1\_2 := & \quad \text{comp.New("comp1\_2", Server)(\text{CP1\_2})} \\
CP2 := & \quad \text{comp.New("comp2", Client)(\text{CP2})} \\
CPP1\_1 := & \quad \text{cpPort.New("cpport1\_1")(\text{CPP1\_1})} \\
CPP1\_2 := & \quad \text{cpPort.New("cpport1\_2")(\text{CPP1\_2})} \\
\end{align*}
\]

AddComp(A1, CP1\_1)
AddComp(A1, CP1\_2)
AddPort(CP1\_1, CPP1\_1)
AddPort(CP1\_2, CPP1\_2)
AddConn(A1, CN1)
AddPort(CN1, CNP1)
AddPort(CN1, CNP2)
Attach(CPP1\_1, CNP1)
Attach(CPP1\_2, CNP2)
We can now specify the pre-conditions that must be met before the refinement schema can be applied, and demonstrate that the initial architecture satisfies these pre-conditions.

3.2.1 Pre-conditions

We can express the preconditions firstly in natural language as: there must exist two distinct components, one of type Server, one of type Client, and they must be connected by a connection of type ClientServerCall. We can express this more formally using the ADTs as follows:

1) \( a \models arch, x, y : \text{comp}; z : \text{conn} \)
2) \( x \neq y \)
3) \( Type(x) = \text{Server} \)
4) \( Type(y) = \text{Client} \)
5) \( Type(z) = \text{ClientServerCall} \)
6) \( Is\text{Connected}(x, y, z) \)

These pre-conditions can be proven by inspection of the ADT representation for the initial architecture. Namely, if we substitute CP1 for x, CP2 for y and CN1 for z we can see relatively easily that the pre-conditions are met. However, as an example, and to show the relative ease of the proofs we can prove pre-condition 3 as follows:

Axiom 2 in the Component ADT (Table 2) states:

\( Type(\text{New}(s, t)) = t \)

Now, from the ADT representation of the initial pattern above we have:

\( CP1 := \text{comp}\text{.New}("\text{comp1}", \text{Server}) \)

Combining the two we get:

\( Type(C1) = Type(\text{New}("\text{comp1}", \text{Server})) = \text{Server} \) (by applying axiom 2)

3.2.2 Post-conditions

We can now also state the post-conditions on the refinement. As with the pre-conditions, we can express the post-conditions in natural language as: there must exist three distinct components, two of type Server and one of type Client. Each of the Server components must be connected to the Client component via distinct connections of type ClientServerCall. We can express this more formally using the ADTs as follows:

1) \( a \models arch, x, x’, y : \text{comp}; z, z’ : \text{conn} \)
2) \( x \neq x’ \neq y \)
3) \( z \neq z’ \)
4) \( Type(x) = \text{Server} \)
5) \( Type(x’) = \text{Server} \)
6) \( Type(y) = \text{Client} \)
7) \( Type(z) = \text{ClientServerCall} \)
8) \( Type(z’) = \text{ClientServerCall} \)
9) \( Is\text{Connected}(x, y, z) \)
10) \( Is\text{Connected}(x’, y, z’) \)

The importance of the pre and post conditions is that in a real design situation, the architect can develop their architectures and then an automated system can use these pre and post conditions to firstly select valid refinements and secondly ensure they are incorporated correctly.

3.2.3 Obligations

Finally, we specify the obligations (using natural language) on the design as a result of using this refinement. Specifically, the obligations are:

• In the final pattern, the functionality of C1_1 and C1_2 must be identical to each other and to the functionality of C1 in the initial pattern
• It must not matter to which out of C1_1 and C1_2 the client makes its request. Hence there is an obligation for the data and services residing on C1_1 and C1_2 to be replicated.

We can also represent this second obligation formally in temporal logic. The essence of this obligation is that once a
user stores a file on the server, the servers must synchronise. Thus, assuming each server has a Store and a Synchronise process, we can state the obligation in temporal logic:

\[ \Box (\text{Server.Store} \Rightarrow \Diamond \text{Server.Synchronise}) \]

The meaning of this temporal logic is that always when Server.Store happens then eventually Server.Synchronise must happen. The eventually operator (at worst) prompts the need to have a NFR value assigned, if it was not in the original specification. This is an example of the adage that Systems Engineering is anything that makes one think about the problem, and stops one buying products and writing code.

4. A Framework for using Non-Functional Refinement Schemas

This section describes a framework for performing architecture-based refinement which focuses on the non-functional requirements while still addressing the functional requirements. The framework is centred around the concept of the refinement schemas introduced in the previous sections.

The framework essentially describes the inputs to the refinement process, the steps and activities that occur during refinement, and the outputs along the way and at the end. It is intended to help the designer as they proceed from abstract to concrete architectural descriptions, bit not to replace the designer altogether. It should help them make the right decisions, force them to stop and ponder their design, and help them avoid common pitfalls. Again, this is the adage that Systems Engineering is anything that makes one think about the problem, and stops one buying products and writing code.

4.1 Details of the Framework

The initial inputs under this framework are the essential architecture and the requirements specification. Before refinement can commence, we need to develop the essential architecture \[10, 31\]. We propose that this essential architecture would typically come from either some sort of architectural style for systems, or from an initial functional partitioning of the system.

A client/server system or a peer-to-peer system are examples of architectural styles and patterns that could be used as the basis of the essential architecture. Alternatively, an example of an essential architecture derived by partitioning key functionality is that of an insurance system, i.e. the computer-based systems that run any insurance company. We have validated with an insurance company, substantiated by IBM \[19\], that the main function of an insurance system is to create policies, handle claims that are made, make payments to claimants, and to issue receipts. Based on this we could propose the essential architecture of the system to be four components (Policies, Claims, Payments and Receipts) interconnected in appropriate ways.

Additionally, before refinement commences, we must have a systems requirements specification (SRS) containing the functional and importantly the non-functional requirements for the system.

During refinement the non-functional refinement schemas, the model checker tool, and non-functional requirements calculators are used \[5, 10\].

The functional requirements are allocated to components within the architecture and must all be allocated before refinement is considered complete. The non-functional requirements are evaluated over the architectures as they are refined. As discussed previously, due to the domain specific nature of the schemas we can employ various refinements in the right context to achieve certain improvements in non-functional qualities. The requirements are checked iteratively, to see if they are allocated, or evaluated to see if they are satisfactory in the case of non-functional requirements.

During refinement whilst using the schemas additional obligations on the design can appear. These can essentially be treated as new requirements that must be fulfilled before the refinement can be considered complete.

After the refinement has finished, all the requirements should be met, and the result is a implementation architecture, or at least an intermediate architecture that represents a more concrete version of the system, which satisfies more of the requirements than the initial essential architecture. \[10\]

The following section presents a worked example following this framework which should serve to further illustrate its use.

5. Worked Example

This section presents a worked example of refining a system using the framework described in the previous section. The system is simple enough to avoid unwarranted complexities for the purposes of this discussion, but sufficient to illustrate the concepts discussed throughout the paper. The specification and requirements for the system are firstly presented, followed by the refinement of the system using the example refinement schema.

5.1 The System

Specification:

A distributed, high-performance, reliable file server system that allows users to store and retrieve files remotely.

Functional requirements:

- Store files
- Retrieve files
Non-Functional requirements:
- Reliability
  - High in File Stores
  - Medium in communications
  - Low in Users
- Performance

5.2 Refinement of the File Server System

The following example forms only a small part of a longer refinement example that would represent the full refinement of this system from the most abstract to the most concrete architecture [11]. Its purpose is to illustrate the potential use of refinement schemas in a design context. Where the example finishes, a real design of the system would continue, undergoing further refinement.

We start the worked example with the essential architecture of the file server system. This sort of system can be described by a general client / server architectural style. Thus, we can present the essential architecture as follows:

![Essential Architecture Diagram]

Given that we are using a general client/server architectural style as the essential architecture, we can represent the essential architecture in the following ADT terms:

```
A := arch.New("essential arch")
CP1 := comp.New("File Stores", Server)
CP2 := comp.New("Users", Client)
CPP1 := cpPort.New("File Stores Port")
CPP2 := cpPort.New("Users Port")
CN1 := conn.New("Request", ClientServerCall)
CNP1 := cnPort.New("Server Port")
CNP2 := cnPort.New("Client Port")
AddComp(A, CP1)
AddComp(A, CP2)
AddPort(CP1, CPP1)
AddPort(CP2, CPP2)
AddConn(A, CN1)
AddPort(CN1, CNP1)
AddPort(CN1, CNP2)
Attach(CPP1, CNP1)
Attach(CPP2, CNP2)
```

Since one of the non-functional requirements for the system is reliability, and the domain of the file server system is, broadly speaking, “client server systems”, we can investigate using the refinement schema presented in section 3.2. The first step is to check whether the pre-conditions (section 3.2.1) are met in the essential architecture.

The proof of the essential architecture satisfying the ADT pre-conditions is similar to the proof done in section 3.2.1 (given that the two architectures are identical aside from naming) and so for the sake of brevity we take it as given that the essential architecture meets the pre-conditions. However, to make it clear why this is the case, we can see that the essential architecture contains a component of type Client (Users), a component of type Server (File Stores), and that the client and the server are connected by a connection of type ClientServerCall (Request). Thus we meet the typing and the topological constraints of the pre-condition and we can propose the new architecture.

![Refined Architecture Diagram]

We must check that we have met the ADT post-conditions of applying the refinement. Again, the proof of this is similar to the proof of section 3.2.1 and so it won’t be repeated. Suffice to say we can see that the refined architecture above contains two components of type Server (File Store 1 and File Store 2) and both are connected by a connection of type ClientServerCall (Request 1 and Request 2) to the component of type Client (Users). Thus the post-conditions are met.

We can now run the reliability evaluator to see how well the current architecture satisfies the reliability requirement. Supposing the reliability evaluator determines that the reliability in the servers is not yet sufficient, and that another server is required, we can apply the refinement again, by the same reasoning as just presented. Doing this results in the following architecture.

![Updated Refined Architecture Diagram]

When we now run the reliability evaluator, it determines that reliability is sufficient, thus triplicating the servers (in this type of system in this domain) has allowed us to meet the reliability non-functional requirement.

We must now pay attention to the obligations put upon us as a result of using the refinement schema. Referring to section 3.2.3 we can see that we (as the designer) must ensure that all the servers have the same functionality, and that the data is replicated across the servers and kept synchronised.

These obligations must be satisfied before the application of this refinement schema can be considered complete. It is reasonable to assume, at this level of abstraction, that the first obligation is met simply by having the three servers of the same component type. That is they are all identical servers with the same functionality. However, even at this level of abstraction the obligation to
ensure data consistency across the servers cannot be considered fulfilled.

The nature of this second obligation suggests that the servers must talk to each other, they must keep their data synchronised. This means that there must be a connection between the servers. As can be seen in the above diagram, one possible route for this communication already exists – via the clients. However, this option can be quickly discarded, especially when considering eventual implementations of the system. Imagine the system distributed across the internet. In this case the servers would have to replicate the data via any or all the clients using the system at any time. What if the clients log off? What if the clients crash? It is obvious that this is not a valid solution.

Rather, we should introduce some communication between the servers. That is, each server should directly “talk” to both of the other servers. This can be represented as follows:

In essence, we have now stepped into a type of “functional refinement” of the system. As designers, we need to satisfy ourselves (in order to satisfy the obligations of the refinement schema) that the functionality exists to replicate the data across all the servers. The current version of the architecture as it exists above does not make this point explicit.

The final refinement in this example makes the data replication functionality explicit. It involves adding sub-components to handle the data replication across the servers. This can be seen in the following diagram:

One final point to make is evident in the statement of the “data replication obligation” in temporal logic (section 3.2.3). We can see that the obligation only states that eventually the data must be replicated. Another output from using this refinement schema is then the further specification of what “eventually” means. Given the requirement for high-performance of the system, we might propose that “eventually” means within 0.5 seconds of the update being received on any computer. When originally specifying the requirements for the system, we didn’t know that we would have to specify such a performance requirement, as we didn’t know that we would replicate the server. It is only from using the refinement schema that these choices and obligations were made apparent.

6. Future Work

Our future work involves many aspects of this paper. Firstly we are investigating augmenting the currently implemented tool with a Prolog or similar based knowledge base for ADT’s and in particular the checking of the pre and post conditions. Additionally we plan to investigate other mathematical formalisms (instead of ADTs, for example first-order logic and various forms of rewriting logic) to see if any them exhibit characteristics that might make the specification and proving of the refinements more tractable for both the practitioner and for tool-based automation.

With regard the tool-based automation, we plan to extend the tool to a point where it can build on its current ability to verify the correctness (in terms of the ADTs) of a given architectural model, to allow it to verify the correctness of a given refinement on a previous architecture based on a knowledge of many refinement schemas.

Additionally, we would like to investigate further the synergies between our refinement schemas and those presented by Barber [5].

Finally, as pointed out in section 2, considerable field work and validation needs to occur for all of these sorts of specification and refinement methods and we plan to undertake some of this work in conjunction with industry partners.

7. Conclusion

We have demonstrated through a simple example, the type of structural refinements that can take place to satisfy certain non-functional requirements. The example illustrates a refinement approach using refinement schemas. The schemas contain pre and post conditions (the proofs of which are tractable for practitioner), the domain in which the refinement is valid, the effect on specific non-functional requirements and finally obligations put unto the design as part of adopting the refinement.

In addition, the worked example shows the cycle of non-functional, to functional, to non-functional refinement occurring. This can be seen as a result of the “trace” or follow on effects from using the refinement schemas. From the simple application of one refinement schema we have refined from a fairly simple architecture to one that is more
concrete and has more meaning within the domain of the system and also satisfies more requirements. Additionally we have elicited further non-functional and functional requirements through the use of the refinement schema.

8. References
