Experiences with using the Systematic Method for Architecture Recovery (SyMAR)

Fritz Solms
Department of Computer Science
University of Pretoria
South Africa
fsolms@cs.up.ac.za

ABSTRACT
This paper provides an introduction to the Systematic Method for software Architecture Recovery (SyMAR) and discusses some experiences in applying the method to an industrial case study.

SyMAR is used to recover a software architecture description which is consistent with the view that software architecture provides the software infrastructure addressing non-functional requirements within which application components addressing functional requirements can be specified, deployed and executed. The resultant architectural description includes the identification of architectural components addressing non-functional concerns, the abstractions into architectural patterns introducing infrastructural constraints, the tactics through which quality attributes are realized, and the description of concepts and constraints which the architecture introduces for application components.

When applying the method to a large industrial case study, it is found that the method does assist in exposing important architectural decisions, but that it is labour-intensive and that it is difficult to assess the completeness of the architectural description produced by using this method.

Categories and Subject Descriptors
D.2.11 [Software Engineering]: Software Architectures

General Terms
Architecture recovery

Keywords
architecture recovery, software architecture, architectural patterns, tactics, software architecture analysis

1. INTRODUCTION
In the context of ongoing development without strict architectural control, it is common for systems to experience architecture erosion and architectural drift [4]. The resultant software architecture might no longer be well understood and documented, resulting in further architecture erosion with potential negative consequences for core architectural qualities like maintainability and reliability. A lack of an authoritative understanding of the software architecture makes it difficult to analyze the causes of architectural concerns and to effectively re-architect aspects of the system to address such concerns [15].

The architecture of a software system is meant to address non-functional requirements including quality requirements and integration requirements. Architecture recovery methods commonly yield components-connector based architectural descriptions without differentiating between architectural components addressing non-functional requirements and application components addressing functional requirements. Abstractions into architectural patterns and tactics are commonly not exposed. Yet it is these abstractions which represent important architectural decisions through which quality attributes are realized.

Software Architecture Recovery refers to the endeavor of recovering or reverse-engineering a software architecture from the available artifacts. It is often a preliminary step which needs to be done in order to obtain an understanding of the architecture of a software system prior to a re-architecting phase during which the software architecture is modified in order to address some architectural concerns. The output of this endeavor is a complete or partial description of the software architecture [5].

Software architecture recovery projects face a number of challenges. Software systems are often complex and may span across millions of lines of code using a wide variety of technologies and frameworks. In addition, the software architecture might use vendor and open-source components which significantly affect the software architecture and there might be only very limited architectural information available for these. This makes it difficult to assess how the quality attributes of the system are affected by including such third-party architectural components within the software architecture.

A further aspect which may complicate architecture recovery is a lack of authoritative resources available for the software architecture recovery. Documentation is typically outdated and the architects who originally architected the system may no longer be available. The current architects and developers may only have an incomplete understanding of the software architecture.

Often there is a level of uncertainty around the deliv-
erables of the software architecture recovery. The type of deliverables depend on one’s definition of software architecture. Currently we have no definitive definition, with competing definitions varying on whether the software architecture is a high-level abstraction of a software system [19], whether it needs to be specified across levels of granularity [3], whether architecture focuses purely on structure or also includes functionality, whether it is a specification which operates at the more abstract level of concepts and constraints within which a software system is designed and developed [6, 14], and whether one can separate software architecture design from the design of application functionality or not [16]. The scope, method and outputs of a software architecture recovery are significantly dependant on one’s choice of software architecture definition.

In this paper we report some initial experiences of using a systematic architecture recovery method which is aligned with the perspective that the “Software architecture is the software infrastructure within which application components providing user functionality can be specified, deployed and executed” [16].

This definition is aligned with the perspective that software architecture addresses non-functional requirements, whilst application components address functional requirements. The application components developed, deployed and executed within the software architecture will provide the application functionality with quality attributes determined by the software architecture. For example, the levels of scalability and reliability at which the application services will be available is largely determined by the software architecture, e.g. whether clustering and resource reuse are used.

Ref. [16] lists the required elements of an architectural specification as

1. the architectural components addressing architectural responsibilities, i.e. responsibilities around addressing non-functional requirements, and the connectors between them,
2. infrastructural constraints in the form of architectural patterns,
3. architectural tactics or strategies through which quality requirements are concretely addressed, and
4. concepts and constraints within which application components are specified and developed, e.g. in service-oriented architectures applications are assembled from standalone, stateless, discoverable services which are connected via pipes from which higher-level services are orchestrated.

2. RELATED WORK

Software architecture recovery involves information extraction of architecturally relevant information, abstraction and the description of the recovered architecture [17]. Many approaches recover a software architecture in the form of components, their interfaces and the connectors between them [17, 5, 8]. One of the benefits of an architecture recovery yielding a component-connector based architecture description is that it can be largely automated [17]. Furthermore, most Architecture Description Languages (ADLs) support the description of a software architecture in terms of components and connectors – see [7] for a survey of ADLs. However, within the component-connector approaches, a core challenge is the grouping of identified components into more abstract architectural components. Dueñas et al. [5] have shown that statistical clustering techniques can be used to assist with the identification of such higher-level, abstract components. They propose an architecture recovery process [5] which iteratively identifies low level elements which are in turn abstracted into higher-level architectural components. The process generates an architectural description in some choice of an ADL showing the components and connectors across levels of granularity.

A further complexity within the component-connector approach to software architecture recovery is the differentiation between architectural components addressing non-functional requirements and application components addressing functional requirements [16]. None of the automated approaches are able to make such a distinction. Consequently automated approaches are also not currently able to recover the concepts and constraints introduced by a software architecture for application components [16].

Pinzger et al. [13] argue that architectural patterns and styles represent important architectural design decisions and that an architecture recovery must expose these abstractions. This view is supported by Heesch et al. [18] who argued that within a component-connector approach to architecture recovery, it is difficult to recover the rationale behind architectural decisions. Using an empirical study, they were able to show that the quality and efficiency of architecture decision recovery is improved by recovering the use of architectural patterns within the software architecture. The deliverable of an architecture recovery is an architectural description. A lot of work has been done around ADLs. However, the attempts at introducing the semantics for describing a software architecture have been largely unsuccessful [7]. It has been more than a decade since the first ADLs have been introduced and yet none of the ADLs are widely used in practice. Part of the reason for this may be that only few ADLs [7] have support for architectural patterns and even less have the notion of architectural tactics i.e. for reusable techniques used to concretely address quality requirements [11]. However, lately we have seen some interesting work which explicitly adds support for the definition of architectural patterns which implement architectural tactics [12]. Pahl et al. [12] define a description-logic based ontology for components, connectors, roles, ports and configurations which introduces pattern-specific concepts and constraints. The ontology can be extended to support the specification of concepts and constraints for application components through pattern-specific component specializations (e.g. pipe and filter components).

3. A SYSTEMATIC ARCHITECTURE RECOVERY METHOD

This section introduces the Systematic Method for Software Architecture Recovery Method (SyMAR). The method is used to recover a software architecture description which is consistent with the view that a software architecture provides the software infrastructure addressing non-functional requirements within which application components addressing functional requirements can be specified, deployed and executed [16].

The resultant architectural description includes
• the identification of architectural components addressing non-functional concerns,
• the abstractions into architectural patterns introducing infrastructural constraints,
• the tactics through which quality attributes are realized, and
• the description of concepts and constraints which the architecture introduces for application components.

Note that within this [16] definition of software architecture, there is a distinction made between architectural components addressing non-functional concerns and application components addressing the functional requirements of the application. Furthermore, both architectural components and application components may exist across levels of granularity.

3.1 Overview of SyMAR

SyMAR (see Figure 1) is an iterative method. Within each iteration the focus is on a specific architectural component. Initially the architectural component is the system as a whole. Subsequent iterations may choose to zoom into a lower level architectural component, choosing one of the sub-components of the previous level of granularity as new subject. But one may also revisit the same or even a higher level component, performing additional extraction, abstraction and description activities in order to refine the architectural description for that component.

The core activities within an iteration can be grouped into

• extraction activities which are used to extract architecturally relevant information from the system,
• abstraction activities to construct and architectural understanding in terms of architectural components addressing architectural concerns and connectors, architectural constraints in the form of architectural patterns and architectural tactics used to concretely address quality requirements, and
• description and presentation which is the process of capturing an architectural description in a form which is consumable by people and/or systems.

3.2 Extraction

It is very inefficient to manually go through the entire source code of a large software system in order to recover its software architecture. Automated recovery approaches using reverse engineering tools may assist with exposing components and connectors and may point out potential grouping of components into higher level components, but they are generally not able to perform the abstraction into architectural patterns and tactics. Furthermore, current tools are not in a position to be able to distinguish between architectural components addressing non-functional requirements and application components which provide the application functionality.

Instead of covering the entire code bulk, one uses a variety of techniques in order to extract information about architecturally relevant components. Examples of such techniques include request-tracing, non-functional concerns analysis, interception analysis (including aspects analysis) as well as third-party component analysis.

3.2.1 Request tracing

In request tracing one traces representative requests or events through the complete processing cycle in order to identify what architectural components are touched. Architectural components are infrastructural components which are concerned with addressing non-functional requirements like accessibility/integrability, scalability, security, reliability, auditability, modifiability and performance. (e.g. quality requirements).

Each architecturally significant component identified during request tracing is tagged and its responsibilities and interconnectivity with other architectural components is analyzed.

3.2.2 Interception analysis

Non-functional concerns are often cross cutting concerns which apply across application functionality. It is for this reason that architectural components addressing non-functional concerns are commonly inserted into the functional processes through either interception or aspects.

A search for interceptors and/or aspects thus often reveals further architectural components which are not exposed by request or event tracing and which were overseen when performing the architectural concerns analysis.

3.2.3 Analysis of architectural impact of third-party components

The software architectures of complex systems typically make extensive use of open source and proprietary frameworks and libraries. Frameworks often provide aspects of the core software architecture and may implement architectural patterns and tactics through which non-functional requirements are addressed. Furthermore, frameworks often introduce concepts and constraints for application components (e.g. enterprise beans and entities in Java-EE).

It is thus important that an architectural description covers any architecturally significant frameworks and libraries within the software system as well as their impact on the quality attributes of the system. Note that the software system might not make use of all architectural tactics implemented by a framework. It is thus important to analyze which of the available tactics are used within the system’s software architecture.

3.3 Abstraction

A core responsibility of architecture recovery is that of abstracting system elements into architectural patterns (or styles) which introduce infrastructural constraints and architectural tactics (or strategies) which are used to concretely realize quality attributes. Also, application components need to be abstracted to identify the concepts and constraints which the architecture introduces to facilitate the development, deployment and execution of application functionality within the architecture.

3.3.1 Abstracting into architectural patterns

Architectural patterns or styles are infrastructural constraints, i.e. they introduce constraints around grouping of components within responsibility domains and connectivity between the components of the different responsibility domains.

For example, the layered architectural pattern groups components within a responsibility domain within separate lay-
ers (e.g. presentation layer, services layer, domain objects layer, . . . ) and requires that components within one layer may only access components which are either within the same or the next lower layer.

The abstraction into architectural pattern thus requires the identification of responsibility domains within which architectural components are grouped and the connectivity constraints between these abstractions.

3.3.2 Architectural tactics

Architectural tactics are used to concretely address quality requirements. For example, resource reuse, clustering, and caching are three examples of strategies used to address scalability.

One of the aims of architecture recovery is to identify the architectural tactics used to concretely address quality requirements. One of the difficulties of this task is that some of the tactics might be implemented in underlying frameworks used by the software architecture, whilst others might be implemented within the system’s code base itself. Furthermore, not all tactics implemented within underlying frameworks are necessarily used. One thus needs to

- identify abstractions of system components into architectural tactics,
- analyze reference architectures and frameworks for the architectural tactics they respectively specify and implement, and
- analyze the use of underlying frameworks in order to assess which of the implemented patterns are used within the target software architecture.

These tasks are non-trivial and difficult to automate. Pahl et al. [12] have introduced an ontology for modeling architectural pattern (styles) which implement tactics addressing quality requirements. This approach is, however, more useful for documenting the use of architectural tactics than for the actual architectural recovery.

To simplify the manual work of discovering architectural tactics used to address quality requirements, software architects commonly maintain, for each quality requirement type (e.g. scalability, security, reliability, . . . ), a list of architectural tactics which can be used to concretely address that quality requirement (e.g. resource pooling, clustering, spreading load across time, encryption, replication, . . . ) [1].

Similarly, software architects maintain lists of tactics which are implemented by frameworks. Being aware of the different tactics widely used to concretely address quality requirements simplifies the task of identifying the use of such tactics during architecture recovery.

3.3.3 Application concepts

A software architecture commonly introduces concepts and constraints within which application components addressing functional requirements are to be developed. For example, a services-oriented architecture introduces concepts like leaf services, composite services, pipes and routers. Higher level services are assembled from lower level services within a pipes and filters paradigm. Java-EE, on the other hand, introduces the concepts of enterprise beans (stateless and stateful session beans, as well as message-driven beans) and entities for the services layers. The Java-EE application concepts for the presentation layer includes facelets and managed beans.

Furthermore, a software architecture generally specifies some constraints which must be adhered to by these application components. For example, a service in SOA must be stateless to facilitate composability of services. and an enterprise bean in Java-EE may not create any threads as this would interfere with the CPU resource management of the application server.

Application components need to be abstracted to capture the concepts and constraints introduced by the soft-

Figure 1: A systematic method for software architecture recovery.
%name

ware architecture for application components which are to be developed, deployed and executed within the software architecture.

3.3.4 Recovering quality attribute trade-offs

Quality requirements are competing. For example, increasing the level of security by increasing the encryption strength may impact negatively on performance, scalability and integrability/accessibility. Software architects aim to align the quality attribute trade-offs done within the architecture with a quality requirements prioritization obtained from the client.

In the context of software architecture recovery one needs to recover One of the aims of abstraction to analyze the employed architectural tactics and patterns in order to recover architectural trade-off decisions which were either explicitly or implicitly made.

3.4 Architecture description

The deliverable of an architecture recovery is an architectural description.

The field of architectural description is still very immature. IEEE 1471-2000 [2] and its latest rework, ISO/IEC 42010 [6]), is an industry standards for an architectural description. It is interesting to note that the definition of software architecture itself has changed across the two versions of this standard [16]. The standard itself specifies requirements which any standards-compliant architectural description must meet. The two fundamental requirements of these standards are that any standards-compliant architectural description must

- be provided in terms of defined views, i.e. each view must comply to a view point specification which introduces the semantics for that view, and
- support the traceability of architectural decisions back to architectural requirements so that the rationale for architectural decisions is captured.

None of the many ADLs are in wide use [7]. Most ADLs support the description of components and connectors. Some do allow for higher-level architectural components to be assembled from lower level components and for the specification of architectural patterns. However, architectural tactics which represent fundamental architectural decisions made to concretely address quality requirements are generally not first-class concepts within the currently available ADLs.

Within industry, one of the more widely used architectural description frameworks is the Kruchten 4+1 View Model of Architecture [10]. This framework is, however, not consistent with the perspective that software architecture is solely concerned around providing an infrastructure which addresses the non-functional requirements within which application components addressing functional requirements can be specified, deployed and executed.

Further work needs to be done around developing an ADL or at least a framework which can be used to describe a software architecture from the perspective of fundamental architectural decisions. Such a framework needs to support the description of architectural patterns and tactics and also needs to provide a way to specify the concepts and constraints the software architecture introduces for application components.

4. CASE STUDY FROM INDUSTRY

Towards the end of 2012, the author was requested to reverse engineer the software architecture of a large banking system and to perform a limited-scope analysis of the software architecture with focus on potential causes of performance and scalability problems. The system is used by corporate banking clients across Africa.

The system was originally developed within a vendor product which was meant to provide the software architecture as well as generic application functionality which could be customized and extended. The hope was that the vendor product would provide a software architecture within which further application functionality can be developed, deployed and executed and that the non-functional requirements would largely be addressed by the vendor solution.

It was, however, found that the vendor-provided software architecture was not able to address the reliability and scalability concerns resulting in a long history of architectural modifications. The end result is that elements of the original vendor solution were absorbed within a Java-EE based home-grown software architecture. The final software architecture has been extensively influenced by services-oriented concepts from the original vendor product and concepts of the Java-EE reference architecture.

The system has around three million lines of code making a manual recovery process with full code coverage impractical and prohibitively expensive.

4.1 Extraction of architecturally relevant information

Due to the size and complexity of the system, it was important to be able to extract the architecturally relevant information without having to go through the entire code bulk. A combination of request tracing, interception analysis and third-party component analysis was used for this purpose.

4.1.1 Request tracing

The result of a single request trace is shown in Figure 2. Request tracing exposed the architectural components used to demarshall and route service requests as well as the persistence infrastructure (not shown here). It did not, however, expose the architectural components addressing security, reliability, scalability and auditability concerns. Note that application logic is only contained in the “business beans” and that all other components are pure architectural components addressing infrastructural concerns and concerns around addressing non-functional requirements.

Note that the Java-EE based software architecture retains a number services oriented elements. Requests are based on document messages which have an envelope containing metadata around the service requested (e.g. the module and function used for routing) as well as a message content containing the core request data. These are separately demarshalled within the routing and service adapter components. Requests are taken through two levels of routing, the first routing the request to the appropriate product and the second routing the request to the appropriate service as offered by a business logic component. In-memory caches for product, component and service handles are used to address performance and scalability concerns. Note that in Java EE thread pooling is achieved through object pooling of business logic components whilst in SOA thread-pooling is at
a service level. The drift from a services-oriented software architecture to a component-based software architecture has resulted in an infrastructure where service handles are maintained as method handles and threads are obtained through component lookup.

Request tracing was done for both real-time requests coming processed from different access channels, application clients and mobile device clients and batch requests submitted through message queues. These traces were used to identify architectural components traversed in the context of request processing.

4.1.2 Interception analysis

Request tracing exposed many of the architectural components around request de-marshalling and routing as well as around persistence. It did not, however, expose how architectural some of the other architectural concerns like security, auditability and reliability were handled.

The reason why request tracing did not expose them was that components addressing these concerns were either inserted at interception points (as was the case for security and reliability) or that the concerns were addressed by the frameworks used (reliability was largely addressed through clustering at the application server and database levels).

Interception analysis did manage to identify a number of architectural elements which were not identified during request tracing. In particular the path into the authorization and logging frameworks was discovered through searching for interceptors.

The authorization infrastructure was largely based on a security interceptor which delegated authorization to the security infrastructure absorbed from the original software architecture provided by a SOA-based Proprietary Banking Framework. The authorization infrastructure allows corporate clients to manage the access rights of their users within the constraints of the services they bought. Authorization validates whether a user can perform the requested function on a target entity.

Similarly, the banking system used an auditability logging interceptor to capture and process events used for auditability logging. It was found that processing around auditability logging was done within the main request processing thread resulting in an impact on the performance and scalability of the system.

4.1.3 Third-party component and framework analysis

The software architecture of the banking system made extensive use the vendor-provided Java-EE framework as well as a SOA-based Proprietary Banking Framework.

The latter provided the SOA-based software architecture for the original system, but in the context of addressing scalability and performance concerns the SOA-based software architecture was replaced by an in-house developed software architecture which made use of the Java-EE reference architecture. Nevertheless, a number of architectural components from the original software architecture were retained. These architectural components continued to provide the software infrastructure for persistence, authorization and auditability logging. These components were absorbed at source level and were analyzed within the source-code based architecture recovery.

Hence the only third-party component which was used as a black box solution was the Java-EE framework. However, not all tactics provided by the framework to address quality requirements were used and even the concepts and constraints for application components were modified somewhat. For example, Java-EE specified stateless and stateful
session beans as application components – i.e. components within which application functionality is specified. Within the software architecture of the banking system these concepts were replaced by the concept of stateless services which were implemented as methods of stateless session beans. The software architecture maintained method handles and introduced services routers which routed requests directly to these stateless methods.

Also, the standard Java-Ee authorization infrastructure implemented within the vendor provided Java-Ee framework was not used. It was replaced with an interceptor which routed authorization requests to the authorization infrastructure absorbed from the original SOA-based banking solution.

4.2 Abstraction

Once the architectural components and their responsibilities were identified an abstraction of the software architecture in terms of architectural patterns and tactics was developed. The concepts and constraints the software architecture introduced for application components was analyzed.

4.2.1 Architectural patterns

At the first level of granularity, the system as a whole was considered as the component. The components were grouped into high-level responsibility domains represented by high-level abstractions. For example, the components in the access/adapter layer all perform request de-marshalling and forwarding to the module delegate for product routing.

```
Client Layer
  Java Swing App
  Mobile App
  <<webServicesClient>> System Client

Access/Adapter Layer
  <<Servlet>> CoreRouter
  <<Servlet>> Java-WS
  <<Message-Driven Bean>> QueueProcessor

Routing Layer
  Module Layer
  ProcessService

Business Layer
  <<StatelessSessionBean>>

Infrastructure Layer
  Persistence Context
  EntityManager
  O/R Mapper

Persistence Layer
  Database
```

Figure 3: The first-level granularity architectural pattern.

The resultant abstractions were found to represent layers within the layered pattern with any layer only having a dependency on the next lower-level layer – i.e. satisfying the infrastructural constraints introduced by the layered architectural pattern. The layers identified in the software system are depicted in Figure 3. Note that only the business logic layer contains components with application functionality.

4.2.2 Architectural tactics

The target architecture of our case study uses a range of architectural tactics to concretely realize required quality attributes. Some of these tactics are provided by the underlying Java-Ee framework, whilst others are implemented within the code of the system itself. The identification of these tactics was done through an analysis of the architectural components identified during the extraction phase together with discussions held with the team which is currently responsible for the software architecture and some senior developers. Such an approach leaves some uncertainty around whether all architectural tactics were identified.

Examples of architectural patterns which are implemented within the code of the target architecture of our case study include caching (of module delegates and method/service handles), interception to apply role-based authorization and auditability logging, as well as queueing and scheduling to spread load across time and improve scalability.

Many of the tactics are implemented within the underlying frameworks used by the software architecture – particularly the vendor provided Java-Ee framework used by the banking system. For example, pooling (including thread pooling, object pooling and connection pooling) and clustering are two of the scalability tactics implemented within the framework.

Not all tactics implemented by the underlying frameworks were used in the target architecture. For example, the authorization tactics specified by Java-Ee and implemented within vendor-provided Java-Ee framework were not used and were replaced by an interceptor which delegated the authorization to components absorbed from a SOA-based Proprietary Banking Framework which validated whether the authenticated principle may perform the requested action on a specific target entity.

4.2.3 Concepts and constraints for application components

The target of our case study is a services oriented architecture which has been evolved into a partially component-based software architecture deployed within a Java-Ee application server. This has resulted in some hybrid concepts within which application components are developed. In particular, application logic is to be provided as stateless services which must be implemented as methods of stateless session beans. Domain objects are implemented as JPA (Java Persistence API) entities which are mapped onto a database structure. Higher-level services are, however, not assembled from lower level services within a pipes and filters paradigm, but instead using the process-manager or controller pattern [9].

4.2.4 High-level view of software architecture

One of the valuable views for software architects is a high-level view which shows the coarse-grained architectural components as they are deployed across hardware nodes. To generate such a view one needs to analyze the deployment
infrastructure. The high-level view of the software architecture of our case study is show in Figure 4.

5. CONCLUSIONS AND OUTLOOK

Using a definition of software architecture which defines the boundary between architectural components and application components and specifies the elements of a software architecture description assisted in defining the scope and deliverables of the architecture recovery. Having a systematic method which guides with the activities used to perform the extraction of architecturally relevant elements and abstraction into architectural elements including components and connectors, patterns, tactics and concepts for application development assisted with stream-lining the process.

There are, however, a number of significant open challenges. The first challenge we faced in the architecture review case study was a way to determine whether the architecture has been fully covered. The code bulk of over 3 million lines of code made it impractical to require full code coverage. Furthermore, large portions of the code focused on implementing application functionality and would not provide further architectural insights.

Another very significant open challenge is that of describing the software architecture. None of the current architecture description languages is in any wide spread use in industry. Furthermore, none was found to be sufficient to support the semantics required for an architectural description which included all the elements to describe the software architecture.

A lot of work has been done in automating aspects of the reverse engineering process. These tools are, however, of limited use as they are generally not able to reverse engineer architectural patterns and tactics. Furthermore, they are not able to distinguish between architectural components introduced to address non-functional requirements and application components encoding processes through which functional requirements are addressed. Generally they project out architectural components as high level abstractions, independent on whether they are concerned around providing application functionality or on addressing quality requirements. The consequence of this is that recovery of an architecture description which specifies the software infrastructure within which application functionality is developed, deployed and executed and which addresses the non-functional requirements of the software system has to be done largely manually. For large, complex systems this is an expensive exercise.

6. ACKNOWLEDGEMENTS

The author would like to thank the University of Pretoria for funding the research.

7. REFERENCES


