Formalising Service-Oriented Design

Mikhail Perepletchikov*, Caspar Ryan, Keith Frampton, and Heinz Schmidt
RMIT University, School of Computer Science and Information Technology, Melbourne, Australia
Email: {mikhailp, caspar, kframpto, hws}@cs.rmit.edu.au

***Abstract*—Service-Oriented Computing (SOC) is an emerging paradigm for developing software systems that employ services. Presently there is already much research effort in the areas of service discovery and orchestration, business process modelling, and the semantic web. While these are all important aspects for moving towards the pervasive adoption of SOC, most existing work assumes the existence of black box services, with little attention given to how such services might be developed in a systematic manner. Furthermore, a precise description of what constitutes a service-oriented system is yet to be formally defined, and the overall impact of service-orientation on the software design process is not well understood. Therefore, this work presents a formal model covering design artefacts in service-oriented systems and their structural and behavioural properties. The model promotes a better understanding of service-oriented design concepts, and in particular, enables the definition of software metrics in an unambiguous, formal manner. Defining such a model and metrics is an initial step towards deriving a comprehensive service-oriented software design methodology.**

***Index Terms*—Service-Oriented Computing (SOC), formal model of software design, metrics, design methodology**

I. INTRODUCTION

Service-Oriented Computing (SOC) is emerging as a promising development paradigm [13, 40] which is based on encapsulating application logic within independent, loosely-coupled stateless services, that interact via messages using standard communication protocols and can be orchestrated using business process languages [5, 30]. The notion of a service is similar to that of a component, in that services, much like components, are independent building blocks that collectively represent an application. However, services are more platform-independent, business-domain oriented, and autonomous and hence decoupled from other services [13] as compared with components.

Service-oriented systems in conjunction with supporting middleware represent Service-Oriented Architecture (SOA), a more abstract concept which is founded on the idea of discovery and orchestration whereby a business process or workflow can identify at runtime the most suitable services for a particular scenario and dynamically compose them in order to satisfy a particular domain requirement [30]. Moreover, in SOA, enterprises should consider services as enablers of business processes that reflect workflows within and between organisations, rather then treating them simply as interfaces to software functionality.

Although SOC (or SOA) is becoming an increasingly popular choice for the development of enterprise software [20], service-oriented (SO) design principles are not well understood and documented, with contradicting definitions and guidelines making it hard for software engineers and developers to work effectively with service-oriented concepts [5, 12, 13, 30]. Consequently, service-oriented systems are often developed in an ad-hoc fashion [14, 22, 42] potentially resulting in lower-quality software being produced.

In contrast, established development paradigms, in particular Object-Orientation (OO), are more widely understood. This understanding is underpinned by formal models of OO design [8], which have facilitated the derivation of systematic development methodologies, and metrics for measuring structural software attributes in order to predict and control the quality of produced software [9]. However, such models, methodologies, and metrics are not immediately applicable to SOC since the design of service-oriented systems is conceptually different to OO, as described in Sections II and III.

Therefore, the main contribution of this paper is the proposition of a formal model (Section IV) covering structural and behavioural properties of the design artefacts in SO systems. This model extends the generic model proposed by Briand et al. [8], where a software design is represented as a relational system, consisting of elements, relations and binary operations. The extensions are based on: i) the core characteristics of service-orientation as described by existing literature [12, 13, 30, 40]; ii) discussions with industry practitioners; iii) analysis of existing SO systems; and iv) the authors' experience with developing SOA-based applications.

The proposed model was designed to be as generic and technology agnostic as possible. Nonetheless, it can be readily customised as shown in Section V where the model was modified in order to allow for the technological constraints imposed by BPEL4WS [4]-based business process scripts.

There are two main benefits of this model. Firstly, it *formalises the design concepts surrounding service-oriented development*, which being an emerging

This paper is based on "A Formal Model of Service-Oriented Design Structure", by M. Perepletchikov, C. Ryan, K. Frampton, and H. Schmidt which appeared in the Proceedings of the 18th Australian Software Engineering Conference (ASWEC'07). ©2007 IEEE.

* Corresponding author

paradigm, have not always been consistently expressed nor well understood [1, 14, 22, 33]. Secondly, this model allows *software metrics related to the structural properties of SO design to be defined and theoretically validated in an unambiguous formal manner* as shown in Section VI . This is of utmost importance since metrics can allow comparison and selection of alternative design structures, as well as detection of quality problems early in the Software Development LifeCycle (SDLC).

Given the above, the model and metrics will *provide* a *foundation for the future development of a systematic service-oriented software design methodology*. The need for the methodology and the outline of the steps required to derive such a methodology are discussed in Section III.

Note that a shorter version of the model was described in a previous publication [35], but the motivation behind its derivation was largely omitted due to the space constraints. In contrast, this article provides a detailed motivation behind this research by outlining the problems related to the lack of systematic guidance for the design of SO systems, and discussing how the proposed model can assist in the derivation of the service-oriented design methodology. Furthermore, it includes additional (cohesion) metrics, and elaborates some of the previously presented model definitions.

The remainder of this paper is organised as follows. Section II describes key characteristics of SOA and SOC. Section III discusses the issues surrounding the development of service-oriented software, thus providing a rationale for this research. Section IV presents a generic model of service-oriented design, which is then specialised in Section V for the specific case of BPEL4WS. Section VI shows example metrics for measuring coupling and cohesion of SO designs that have been defined based on the proposed model. Section VII describes related work on the modelling of software designs and architectures. Finally, Section VIII closes with conclusions and a discussion of future work.

II. FUNDAMENTAL PRINCIPLES OF SERVICE-ORIENTATION

This section describes the key characteristics of the service-oriented paradigm that distinguish it from the Procedural and OO development. The description includes definitions of SOA and SOC, as well as a summary of the key principles of service-orientation that influence the overall design process.

Service-Oriented Architecture (SOA) represents an abstract model of software systems that employ services, which are composed and orchestrated to fulfil a specific domain or business requirement. SOA involves two primary parties: service providers, who publish service descriptions and realise such services; and service consumers, who locate a service description, generally via a registry, and invoke a service [12, 14].

One of the main advantages of SOA is its business alignment [6, 31]. In contrast to the traditional approach of embedding business logic within the software code itself, SOA utilises independent executable business processes represented in terms of business concepts rather than system level implementation details. Such processes can be designed by business modellers with the aid of software tool support and then transformed into executable modules or business process scripts, which are deployed and executed using middleware.

While SOA represents a conceptual architecture of service-oriented systems without enforcing any constraints on the design and implementation of individual services, Service-Oriented Computing (SOC) is the software development paradigm based on the concept of encapsulating application logic within loosely coupled, stateless services that interact via messages [30]. Services in SOC are usually self-contained and should not depend on the context or state of other services [14, 40], thereby allowing high reusability.

Note that although services are somewhat similar to components in Component-Based Development (CBD), implementation inheritance and its complications [39] are not present in SOA. Also, services can be implemented using a range of different technologies and development paradigms. Therefore, service implementation elements should be treated as separate units based on their underlying technology. For example, business process scripts should be considered as a separate implementation element type as they have characteristics that differentiate them from Procedural or OO implementation elements. Consequently, the concept of a service implementation element or module can be sub-divided into: business process scripts, OO classes, and procedural packages (collection of procedures). This is captured by the model described in Section IV.

Also, an interface should be a first class design artefact in service-oriented systems which are based on loosely coupled services. For example, one advantage of using SOAP-based web services, is that they are accessed through a language and location independent interface thus promoting loose coupling [3]. Furthermore, if we consider the use of interfaces in OO languages (which are often used to implement services), coupling through interfaces again has a lower coupling than through class types, as widely advocated by experienced practitioners [18]. Thus, OO interfaces and Procedural headers will also be treated as separate entities in the proposed model.

Given the differences described above, the following key principles of service-orientation have a great influence on the software design process, and as such, will be reflected in the model described in Section IV:

- **Building for reuse is a key design principle of SOC**

Service-oriented systems should consist of independent, platform and language agnostic services that exhibit high reuse due to low coupling [14, 40]. As such, services should be made as autonomous as possible and *inter-service communication should be performed only via service interfaces*.

- **SOC introduces an extra level of abstraction**

The procedural paradigm has only one level of abstraction – the *procedure*. OO has two levels of abstraction: *methods* aggregated into *classes*. SOC introduces a third level of abstraction and encapsulation - a service, where *operations* (procedures/method, etc) are

aggregated into *elements* (classes/procedural packages, etc.) that implement functionality of a *service* as exposed through its service interface.

- **Services can be implemented by elements belonging to various development paradigms / languages**

This is especially relevant given the application of SOC to integration projects with existing/legacy systems. Previous research has shown that the use of different development paradigms, such as Procedural design and OO, will result in systems with different structural properties [11]. This may be exacerbated when different development paradigms are combined, for example an OO system accessing legacy procedural code as would be the case if C code was called via a Java Native Interface.

- **Correctly identifying service interfaces is one of the most important service-oriented design activities**

This is due to the fact that interface granularity and cohesiveness of service operations will strongly influence the overall structure of elements implementing this service. Also, service interfaces should be designed as the primary internal and external entry points of a system; thus they must be highly stable as changes to the interface can potentially influence a large number of clients.

- **Business logic should be encapsulated within independent business processes**

Enterprises depend on information technology for their everyday tasks, therefore business logic and rules are an essential part of modern software systems. The traditional approach is to code business logic into the software itself, whereas SOA advocates situating business logic within centralised business processes that can be easily designed and modified by business modellers with the aid of software tool support [31]. This allows encapsulating business logic at a higher level of abstraction, and also results in the increased reusability of individual services.

- **A service is not an explicit design construct**

This is in contrast to OO design, in which a method is physically encapsulated within a class, whereas a *service boundary is logical rather than physical* in existing implementation technologies. As such, there is a need to establish the guidelines for the allocation of implementation elements to services in an unambiguous formal manner, thereby drawing a physical service boundary and allowing specification of design metrics.

III. Towards a Service-Oriented Software Design Methodology

This section describes major issues related to the development of service-oriented software, with sub-section III.A arguing the need for the SOA-specific design methodology, and sub-section III.B presenting an outline of steps required to derive such a methodology. This provides a motivation for this research since formalising SO design structure should be a first major activity when deriving a design methodology.

A. Rationale

To date, service-oriented design concepts, especially at the implementation level, are not fully understood. Consequently, service-oriented systems are often developed in an ad-hoc manner with little consideration given to good software engineering techniques and practices [14, 22, 42]. This is unfortunate since the correct engineering of software has obvious advantages in terms of the overall development process and software quality in any development paradigm, whether it is a well-established one such as OO [38] or an emerging paradigm such as Agent-Oriented Development [29].

Furthermore, rigorous software engineering is especially significant for the emerging generation of service-oriented systems due to their large scale, complexity, and data volume [14]. Additionally, such systems should accommodate rapid business changes, therefore they should be designed not only for efficient and reliable performance, but also for maintainability which is one of the most-resource consuming activities in the whole Software Development LifeCycle [41].

Moreover, the importance of systematic software design methodology will increase as service-oriented systems are becoming larger, more heterogeneous, and are developed by teams that are as heterogeneous and geographically dispersed as the systems they seek to build. Designing and maintaining such systems can be a daunting task, to which systematic software methodology can bring formal and clearly specified processes and guidelines. However, in much the same way that the OO paradigm required different methodologies [38] than the Procedural paradigm that preceded it, so too does SOC differ from OO. Not only is this intuitively clear, since SOC has aspects that are both procedural and object based, it also has additional characteristics that do not fit neatly into either of the aforementioned paradigms as discussed in the previous section.

Presently, even at the high level there are conflicting opinions as to which general strategy should be used when developing SOA-based systems [5, 23]. For example, according to Fowler et al. [17], developers should not try to design an application into disparate web services that talk to each other. Rather, they should build the application and expose various parts of it as web services, treating those web services as remote facades. In contrast, Barry [6] and Singh et al. [40] believe that simply adding web services to an existing application does not constitute SOA. They argue that the system should be composed of discrete internal and external services resulting in a loosely coupled system. Consequently, developers following this approach would structure the entire system as a set of interacting services, from the very start of the development life-cycle. The former view advocates a bottom-up approach, where developers build the application, add web services to it, and then combine services into business processes. Conversely, the latter view prescribes a top-down approach based on business domain decomposition.

Finally, little research effort has been dedicated to considering how the structural properties of service-oriented designs (for example coupling and cohesion) may impact non-functional software quality attributes such as maintainability, efficiency, reliability, and security. The correlation between structural and non-

functional attributes (measured using software metrics) has been shown in previous paradigms [9, 19], and therefore needs to be investigated for the SO paradigm.

B. Methodology Outline

The above has underlined the importance of creating a software design methodology for SOA. Such a methodology should incorporate a detailed and precise specification of models, metrics, notations, processes, and guidelines. Developing this methodology will involve many aspects including formal modelling, mathematical analysis, case studies and empirical design studies. The following summarises major steps required to derive a service-oriented software design methodology:

- **Defining a formal model of service-oriented system design structure**

Formalising the structure of SO systems should be the first step towards defining a design methodology. A formal model of SO software design will promote a better initial understanding of the service-oriented paradigm by encapsulating the major structural and behavioural properties of design artefacts (services, service interfaces, service implementation elements, etc.) in service-oriented systems. Moreover, the model will allow the precise definition and specification of metrics for measuring structural design properties; as well as the establishment of predictive models for estimating non-functional quality characteristics of service-oriented software.

- **Defining and validating metrics for quantifying service-oriented design structures**

The design-level metrics will lay a foundation for the design methodology. This is because metrics can be used to measure and evaluate structural design properties, thereby allowing identification and prevention of design problems. The metrics can also be used to predict, and therefore manage, some of the non-functional quality characteristics of software products (such as maintainability) early in the SDLC. This is important since the methodology should incorporate guidelines and best practices that target both functional and non-functional software requirements and characteristics.

Given the formal model of SO designs, the definition and theoretical validation of metrics can be done in a precise mathematical manner with the overall process characterised by the following activities: i) establish measurement goals (e.g. measure coupling between SO software artefacts in order to predict maintainability); ii) set experimental hypotheses that will be used in the empirical validation of metrics; iii) formally specify metrics using the definitions from the proposed model; iv) theoretically validate metrics using the definitions from the proposed model; v) empirically validate and, if necessary, refine metrics.

- **Specifying service-oriented design methodology**

Specifying the methodology should be an iterative process consisting of ongoing evaluation and refinement. To this end, the metrics can provide the first means for identifying initial design guidelines and patterns. For example, the metrics can be ranked in terms of their impact on the non-functional attributes. This is necessary since, as an example, 17 different coupling metrics and 6 cohesion metrics were defined in previous work [34, 36] and thus it is important to determine those which have the greatest positive or negative effect so that they can serve as design patterns and anti-patterns respectively.

Furthermore, following comprehensive empirical validation of the metrics, more specific design guidelines can be formulated in terms of concrete metric values. For example, minimum and maximum recommended degrees of coupling for the most important services in the system can be suggested based on the data collected from existing projects. Another important resource for assisting in the derivation of methodological guidelines and design patterns will be the knowledge of experienced service-oriented designers, developers and architects, which should be elicited through the analysis of data collected from qualitative and quantitative studies.

- **Empirical studies and methodology evaluation**

Since the development of the methodology is by necessity analytical, empirical studies should be conducted in order to provide a rigorous evaluation of the methodology. Such studies should involve participants with varying levels of experience in SOA/web services development performing design tasks, with the independent variable being specific methodological approaches and techniques, and the dependent variables the various non-functional attributes (such as maintainability), which can be predicted using structural design metrics. This shows another application of the metrics, in which they will be used to assist in the empirical evaluation of the methodology. Finally, established experimental techniques for creating unambiguous design tasks, collecting data, and statistically analysing the results should be used. Such experiments will provide the basis for the iterative refinement and enhancement of the methodology.

- **Developing automated tool support**

Given that modern software systems comprise a large number of artefacts that are shared among a range of personnel with various roles and expertise, automation and tool support is crucial for tasks that are systematic or repetitive. For example, automated tool support should be provided for: i) editing architectural and detailed design diagrams using graphical representation of the formal definitions specified in the model presented in the Section IV; ii) the automatic collection of structural design metrics from the design artefacts; iii) simulation tool support for generating usage information and environmental or deployment context profiles; and the automatic generation of efficiency and reliability metrics.

To conclude, defining a formal mathematical model covering the structural and behavioural properties of SO system design should be a first step towards defining a comprehensive systematic design methodology. In addition to encapsulating the key characteristics of service-orientation to support a better understanding of the issues related to SO development, the model will provide means for defining structural software metrics in an unambiguous formal manner. Such metrics can be readily validated theoretically using one of the well established set-based validation approaches (such as the

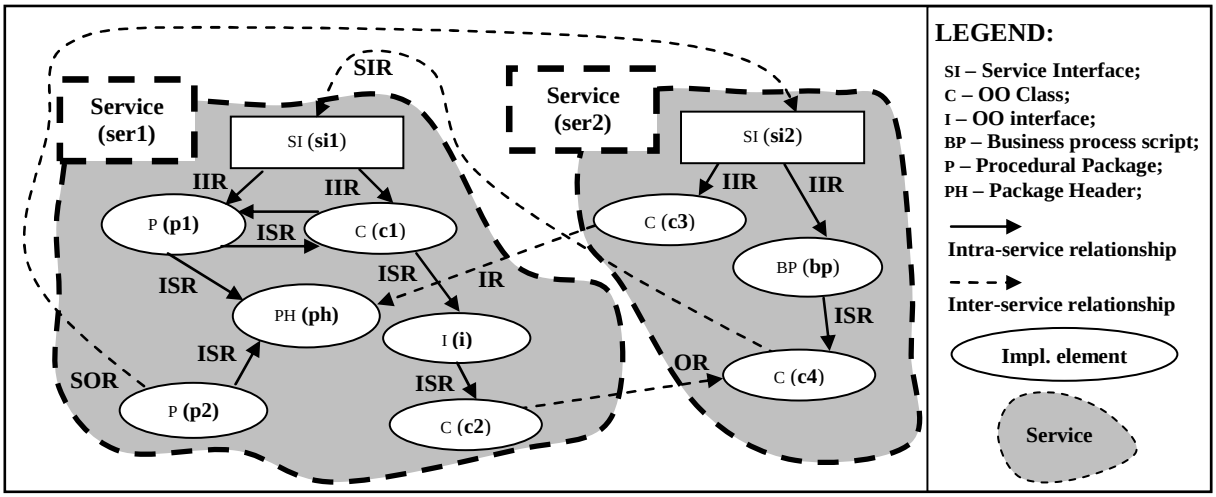

Figure 1. Example design of service-oriented system (SOS)

property-based software engineering measurement framework [8]) since the model uses set theory to capture the design of SO software. Finally, the model will serve as the basis for automatic design consistency checks and metric collection, therefore supporting identification of design flaws.

IV. Formal Model of Service-Oriented Design

This section presents a formal model of service-oriented system design that was derived in order to support the formal definition of metrics, thus providing initial guidance for the derivation of the methodology as described in sub-section III.B.

The model extends the generic model proposed by Briand et al. [8] in order to *capture the design structure of a service-oriented system as a bi-directional graph expressed using set-theoretic notation.* In this model, a software system *S* is represented as a pair $<E, R>$[1], where *E* represents the set of elements of *S*, and *R* is a binary relation on *E* ($R \subseteq E \times E$) representing the relationships between the elements of *S*. Vertices in the graph symbolise design artefacts (the set of elements *E*) representing logical and physical software entities found in service-oriented systems, namely service interfaces and various implementation elements. Edges correspond to the relationships between these design artefacts (the set of relationships *R*), representing both structural and behavioural dependencies. Finally, from the graph of a service-oriented system, specific sub-graphs representing individual services can be derived.

Fig. 1 shows an example graph representing the arbitrary design structure of a service-oriented system with two marked sub-graphs representing its constituent services. As described above, this graph can be captured using conventional graph theory notation [16] being represented as a vertex set and an edge set, where an edge with end vertices *x* and *y* is denoted by *(x, y)*. For example, a system SOS (Fig. 1) consists of:

- vertex set V(SOS) = {si1, si2, p1, c1 c3, ph, i, bp, p2, c2, c4};
- edge set E(SOS) = {(si1, p1), (si1, c1), (c1, p1), (p1, c1), (c1, i), (i, c2), (p1, ph), (p2, ph), (p2, si2), (c3, ph), (si2, c3), (si2, bp), (bp, c4), (c4, si1)}.

Additionally, Fig. 1 shows two sub-graphs of *SOS*, which represent services *ser1* and *ser2*. For example, service *ser1* consists of a vertex set *V(ser1)* which is a subset of vertex set *V(SOS)*. This can be expressed as *V(ser1)* = {si1, p1, c1, ph, i, p2, c2}; and an edge set *E(ser1)* = {(si1, p1), (si1, c1), (c1, p1), (p1, c1), (c1, i), (i, c2), (p1, ph), (p2, ph)}.

Note that one of the key aims of the model is to identify various intra- and inter-service relationships given that a service is not an explicit design construct in existing implementation technologies, i.e. a service boundary is logical rather than physical (this is in contrast to, for example, an OO method which is physically encapsulated within an OO class). Therefore, the allocation of implementation elements to services was done by considering the possible call paths in response to invocations of service operations via the service interface. In practice, this information can be derived from behavioural design artefacts such as sequence or collaboration diagrams, and flow charts; or by tracing code execution where available.

As an example, consider again the sample design of Fig. 1. Even though class *c2* is statically coupled to class *c4*, the element (class) *c4* is not considered to be part of service *ser1* since it is assumed that this element is not reachable through methods invoked on *c2* through operations on interface *si1*. However, in the case of service *ser2*, this element is included since it is assumed that *c4* is now invoked by business process script *bp* in response to the invocation of some of the operations of service interface *si2*.

Also, as stated previously the decision was made to create a generic model that was independent of technology. For example, an OO class design entity is a general concept independent of a particular implementation language such as Java or C++. Furthermore for generality, the structural characteristics of a generic business process script are equivalent to those of a procedural package.

[1] Refer to Appendix A for the notation used in the model definitions

Similarly, for generality, the concept of a separate aggregate software component such as an Enterprise Java Bean (EJB) or CORBA component is not included in the model as it can be represented as a combination of interfaces and classes. The same applies to other popular technologies such as scripting languages, since they can be classified as OO/Procedural implementations.

Finally, due to the size and complexity of the model it is presented in stages to improve readability, with i) sub-section IV.A describing the entities representing design artefacts; ii) sub-section IV.B describing the relationships between such entities; iii) sub-section IV.C providing a complete model that combines the definitions from the former two subsections; and finally iv) sub-section IV.D presenting different types of service-oriented systems.

A. System Structure

This subsection defines the structure of a service oriented system in terms of its services and constituent design entities. The relationships between these entities will be described in the following sub-section.

□ DEFINITION 1 (System structure, Service structure)
The sets representing the compositional elements of a service (*s*) are subsets of the sets comprising the total elements of the service-oriented system (*SYS*), with the exception of the service interface (si_s) which is a single element because a service logically has only a single service interface. Formally,

$$\boldsymbol{SYS} = \langle SI, BPS, C, I, P, H \rangle$$

where SI is a set of all service interfaces in the system; BPS is a set of all business process scripts; C is a set of all OO classes; I is a set of all OO interfaces; P is a set of all procedural packages; and H is a set of all package headers.
Given a system (*SYS*), a service ***s*** can be defined as:

$$\boldsymbol{s} = \langle si_s, BPS_s, C_s, I_s, P_s, H_s \rangle$$

is a service of SYS if and only if $si_s \in SI \wedge (BPS_s \subseteq BPS \wedge C_s \subseteq C \wedge I_s \subseteq I \wedge P_s \subseteq P \wedge H_s \subseteq H) \wedge (BPS_s \bigcup C_s \bigcup I_s \bigcup P_s \bigcup H_s \langle\rangle s)$. Note that ⟨⟩ symbol represents *service membership*. As was explained previously, a service boundary is logical rather than physical, thus we need to examine the possible call paths in response to invocations of service operations via the service interface in order to determine whether an element is a member of a service. Formally, an element *e* is a member of a service *s* only if *e* belongs to collaboration *c* (refer to Definitions 5.A and 5.B) of si_s i.e. if *e* belongs to some collaboration sequence $cs \in CS$ as part of collaboration $c_{so} \in CO(si_s) = \langle Param(so \in SO(si_s)), CS \rangle$ (refer to Definitions 3 and 5).

Also note that some of the artefacts could be absent from the system/service structure (e.g. a service could have an OO implementation with no procedural packages). As such, the corresponding sets of elements would be empty as indicated by ∅, but the above definitions would still hold. For example, the following is the representation of a service-oriented system *SOS* and a service *ser1* from Fig.1:
SOS = <SI, BPS, C, I, P, H> = <{si1, si2}, {bp}, {c1, c2, c3, c4}, {i}, {p1, p2}, {ph}>; *ser1* = $\langle si_{ser1}, BPS_{ser1}, C_{ser1}, I_{ser1}, P_{ser1}, H_{ser1} \rangle$ = <si1, ∅, {c1, c2}, {i}, {p1, p2}, {ph}>

□ DEFINITION 2 (Operations of an element)
All implementation elements have collections of callable operations, which can be treated generically for all implementation element types. Formally,

For each element $e \in SI \cup BPS \cup C \cup I \cup P \cup H$ let $\boldsymbol{O(e)}$ be the set of generic operations *o* of element *e*

In addition, specific operations can be defined for different element types, e.g. operations included in a service interface can be defined as: For each service interface $si_s \in SI$ *let* $SO(si_s)$ be the set of service operations *so* of service interface si_s.

□ DEFINITION 3 (Operation parameters)
All operations have sets of parameters. As was the case with operations these can be treated generically as follows:

For each operation $o \in O(e)$ let $\boldsymbol{Param(o)}$ be the set of generic parameters of *o*

In addition, parameters can be defined for different operation types, e.g. parameters to the service interface operations can be defined as: For each service operation $so \in SO(si_s)$ *let* $Param(so)$ be the set of parameters of *so*.

B. Relationships

This subsection describes the coupling relationships between service-oriented design elements. The individual relationships between directly coupled elements are expressed in Definition 4. Next, relationships are considered within the context of services in Definitions 4.A-4.D. Finally, the runtime collaboration between multiple elements during a specific invocation is described in Definitions 4.E and 4.F.

Note that since this paper aims to present a generic model, a general definition of a relationship between implementation elements is proposed as follows:

> A relationship exists between two elements *a* and *b* if an operation in *a* either declares an instance of *b* (including as parameters or return types of an operation) or references an operation or variable implemented in *b* (including through inheritance or other derivation mechanisms). Furthermore, if *b* is also related to *a* according to the above, this is considered to be a separate relationship.

□ DEFINITION 4 (relationships between different design artefacts in service oriented systems)
Since not all combinations of relationships are possible, and to assist in describing the intention behind expressing the different relationships between differing element types, the relationships are described as follows:

Firstly, a set of *common* relationships R_c represents relationships that are expected to occur in all service-oriented systems, in which collaboration between software elements is done either through a service interface or between elements of the same development paradigm. For example a class invoking another class directly (C×C) or through an interface which is implemented by a specific class (C×I). Formally,

$$\mathbf{R_c} = \langle CSI \cup SIC \cup CC \cup CI \cup IC \cup II \cup PSI \cup SIP \cup PP \cup PH \cup HH \cup BPSSI \cup SIBPS \cup BPSBPS \rangle$$

where CSI ⊆ C × SI, SIC ⊆ SI × C, CC ⊆ C × C, CI ⊆ C × I, IC ⊆ I × C, II ⊆ I × I, PSI ⊆ P × SI, SIP ⊆ SI × P, PP ⊆ P × P, PH ⊆ P × H, HH ⊆ H × H, BPSSI ⊆ BPS × SI, SIBPS ⊆ SI × BPS, BPSBPS ⊆ BPS × BPS. For example, a set of relationships CI representing subset of all OO classes to OO interfaces relationships (C × I) for system *SOS* would be CI ={*(c1, i)*} in the design shown in Fig. 1, where each single relationship is represented as an *ordered pair (source, destination).*

Secondly, a set of *possible* relationships R_p is specified comprising those relationships which are technology dependent, insofar as elements collaborate with other elements of a different development paradigm. For example a function within a procedural package is called from a method of a class via a native interface. Formally,

$$\mathbf{R_p} = <CP \cup PC \cup CH \cup CBPS \cup BPSC \cup BPSI \cup PBPS \cup BPSP \cup BPSH \cup PI >$$

where CP ⊆ C × P, PC ⊆ P × C, CH ⊆ C × H, CBPS ⊆ C × BPS, BPSC ⊆ BPS × C, BPSI ⊆ BPS × I, PBPS ⊆ P × BPS, BPSP ⊆ BPS × P, BPSH ⊆ BPS× H, PI ⊆ P × I

Finally, some relationships are considered to be *impossible* within the logical and technological constraints of a service-oriented system. As an example a WSDL based service interface (si∈ SI) cannot call another service interface (or other explicit interface types such as OO interface (i∈I) or Procedural header (h∈H)) directly since this would be done through a separate implementation element such as a business process script or code module. Also, a Procedural header (h) can be coupled to other headers only (through "includes" relationships), it cannot be coupled directly to other elements. Finally, it is impossible to have a relationship from an OO interface (i) to the elements belonging to different paradigms such as Procedural packages (P) and headers (H), and Business Process Scripts (BPS). For completeness these are listed below.

$$\mathbf{R_i} = <SISI \cup SII \cup ISI \cup SIH \cup HSI \cup HP \cup HC \cup HI \cup HBPS \cup IH \cup IP \cup IBPS >$$

where SISI ⊆ SI×SI, SII ⊆ SI×I, ISI ⊆ I×SI, SIH ⊆ SI×H, HSI ⊆ H × SI, HP ⊆ H × P, HC ⊆ H ×C, HI ⊆ H × I, HBPS ⊆ H×BPS, IH ⊆ I× HI, IP ⊆ I × P, IBPS ⊆ I × BPS

Note that these sets of relationships are based on current practice in software-engineering community [12] and the experience of the authors, but are not considered definitive and could change in response to changing technology. Nonetheless, they can be readily used in order to check the consistency of SO software designs.

The set of overall coupling relationships *R* in a service-oriented design can therefore be represented as a union of all *common* and *possible* relationships $\boldsymbol{R = R_c \cup R_p}$. Accordingly, the relationships belonging to a particular service *s* can be represented as: $\boldsymbol{R_s = R_{cs} \cup R_{ps}}$

The following definitions address relationships involving one or more services. Defining such relationships is important since they encapsulate some of the key principles of service-orientation described in Section II. For example, the IR and OR relationships specified in Definition 4.C should be avoided in order to conform to the "building for reuse" design principle of SOC (refer to Section II). In contrast, the SIR and SOR relationships (Definition 4.D) should be encouraged since they ensure that "service interfaces are the primary internal and external entry points of a system".

□ DEFINITION 4.A (relationships between a service interface and service implementation elements)

The set of direct interface to implementation relationships *IIR(s)*, which represents the relationships between a service interface si_s and the implementation elements *e* of service *s* which implements si_s, is defined as follows:

$$\boldsymbol{IIR(s)} = \{(si_s, e) \in R_s \mid R_s \subseteq (SIBPS \cup SIC \cup SIP) \wedge si_s \in SI \wedge e \in (BPS_s \cup C_s \cup P_s)\}$$

An example of IIR relationships set for service *ser1* from Fig. 1 is as follows*:* IIR (*ser1*) = {(si1, c1), (si1, p1)}. Note that as previously described in Definition 4, a service interface cannot be connected to other types of explicit interfaces due to current technological constraints.

□ DEFINITION 4.B (relationships between service implementation elements)

The set of internal service relationships *ISR(s)*, which represents the interconnection of implementation elements e_1 and e_2 belonging to the same service *s* is defined as follows:

$$\boldsymbol{ISR(s)} = \{(e_1, e_2) \in R_s \mid R_s \subseteq (CC \cup CI \cup IC \cup II \cup PP \cup PH \cup HH \cup BPSBPS \cup CP \cup PC \cup CH \cup CBPS \cup BPSC \cup BPSI \cup PBPS \cup BPSP \cup BPSH \cup PI) \wedge e_1, e_2 \in (BPS_s \cup C_s \cup I_s \cup P_s \cup H_s)\}.$$

For example, ISR (*ser1*) = {(c1, p1), (p1, c1), (c1, i), (i, c2), (p2, ph)} in Fig. 1.

□ DEFINITION 4.C (relationships between the service implementation elements of a given service and the elements belonging to the rest of the system)

The implementation elements e_1 belonging to a particular service *s* are connected to the implementation elements e_2 belonging to the rest of the system by:

i) incoming relationships (*IR*):

$$\boldsymbol{IR(s)} = \{(e_1, e_2) \in R_s \mid R_s \subseteq (CC \cup CI \cup IC \cup II \cup PP \cup PH \cup HH \cup BPSBPS \cup CP \cup PC \cup CH \cup CBPS \cup BPSC \cup BPSI \cup PBPS \cup BPSP \cup BPSH \cup PI) \wedge e_1 \in (BPS\text{-}BPS_s \cup C\text{-}C_s \cup I - I_s \cup P - P_s \cup H - H_s) \wedge e_2 \in (BPS_s \cup C_s \cup I_s \cup P_s \cup H_s)\}$$

ii) outgoing relationships (*OR*):

$$\boldsymbol{OR(s)} = \{(e_1, e_2) \in R_s \mid R_s \subseteq (CC \cup CI \cup IC \cup II \cup PP \cup PH \cup HH \cup BPSBPS \cup CP \cup PC \cup CH \cup CBPS \cup BPSC \cup BPSI \cup PBPS \cup BPSP \cup BPSH \cup PI) \wedge e_1 \in (BPS_s \cup C_s \cup I_s \cup P_s \cup H_s) \wedge e_2 \in (BPS\text{-}BPS_s \cup C\text{-}C_s \cup I - I_s \cup P - P_s \cup H - H_s)\}$$

For example, IR (*ser1*) = {(c3, ph)}; and OR (*ser1*) = {(c2, c4)} in Fig. 1.

□ DEFINITION 4.D (relationships between service implementation elements of a particular service and other service interfaces)

The implementation elements *e* of a service *s* are connected to other services in the system (strictly through service interfaces *si*) by:

i) service incoming relationships (*SIR*):

$$\boldsymbol{SIR(s)} = \{(e, si) \in R_s \mid R_s \subseteq (BPSSI \cup CSI \cup PSI) \wedge e \in (BPS - BPS_s \cup C - C_s \cup P - P_s) \wedge si = si_s \wedge si \in SI\}$$

ii) service outgoing relationships (*SOR*):

$$\textbf{\textit{SOR(s)}} = \{(e, si) \in R_s \mid R_s \subseteq (BPSSI \cup CSI \cup PSI) \wedge e \in (BPS_s \cup C_s \cup P_s) \wedge si \neq si_s \wedge si \in SI\}$$

For example, SIR (*ser1*) = {(c4, si1)}; and SOR (*ser1*) = {(p2, si2)} in Fig. 1.

□ DEFINITION 4.E (direct collaboration relationships between service-oriented design entities)

To capture the dynamic aspects of service structures, a concept of a *direct collaboration* (***c_o***) was introduced. A collaboration c_o captures elements that interact in order to achieve some desired functionality in response to *all possible* invocations of operation *o* belonging to some element *e*. Formally:

$$\textbf{\textit{}}c_o \in CO(e) = \langle Param(o \in O(e)), CS \rangle$$

where Param(o∈O(e)) represents parameters (inputs) to the operation *o* belonging to set of operations *O(e)* of element *e* as per Definitions 2 and 3; *CO(e)* is a set of all collaborations of element *e*; and *CS* is the set of *collaboration sequences* (***$cs_{o \in O(e)}$***). A collaboration sequence captures the set of interacting elements that achieve functionality exposed in operation *o* based on *specific inputs* and can be defined as:

$$cs_{o \in O(e)} \in CS(e) = \langle SI_{cs}, BPS_{cs}, C_{cs}, I_{cs}, P_{cs}, H_{cs} \rangle$$

where $SI_{cs} \subseteq SI$, $BPS_{cs} \subseteq BPS$, $C_{cs} \subseteq C$, $I_{cs} \subseteq I$, $P_{cs} \subseteq P$, $H_{cs} \subseteq H$. This represents the set of interacting elements that achieve functionality exposed in operation *o* based on specific inputs. In terms of graph theory notation [16], collaboration sequence $cs_{o \in O(e)}$ represents an open or closed walk starting at element *e*.

□ DEFINITION 4.F (indirect collaboration relationships between service-oriented design entities)

Additionally, a concept of an *indirect collaboration* (***$ic_o \in ICO(e)$***) was introduced in order to capture the *indirect collaboration sequences* (***$ics_{o \in O(e)} \in ICS(e)$***) that include indirectly connected elements determined based on the overall static coupling (disregarding whether the elements are interacting strictly to achieve desired functionality). Note that the definitions of *ic*, *ICO*, *ics*, and *ICS* are the same as the ones for *c*, *CO*, *cs*, and *CS* only the rules for assigning the elements to collaborations are different.

C. Combined Structure and Relationships

This section presents a complete model combining the structure and relationships from Definitions 1-5.

□ DEFINITION 5 (Service-Oriented System and Service)

In the general case, a service-oriented system *SOS* is formally defined as:

$$\textbf{\textit{SOS}} = \langle SI, BPS, C, I, P, H, R \rangle$$

Given a system (*SOS*), a service *ser* is defined as:

$$\textbf{\textit{ser}} = \langle si_{ser}, BPS_{ser}, C_{ser}, I_{ser}, P_{ser}, H_{ser}, R_{ser} \rangle$$

Based on the above definitions, the *inclusion*, *union* and *intersection* operations can be defined as follows:

- ***Inclusion***: service $\boldsymbol{s} = \langle si_s, BPS_s, C_s, I_s, P_s, H_s, R_s \rangle$ is said to be included in service $\boldsymbol{t} = \langle si_t, BPS_t, C_t, I_t, P_t, H_t, R_t \rangle$ (notation $\mathbf{s \subseteq t}$) if $BPS_s \subseteq BPS_t \wedge C_s \subseteq C_t \wedge I_s \subseteq I_t \wedge P_s \subseteq P_t \wedge H_s \subseteq H_t \wedge R_s \subseteq R_t$

- ***Union:*** The union of services $\boldsymbol{s} = \langle si_s, BPS_s, C_s, I_s, P_s, H_s, R_s \rangle$ and $\boldsymbol{t} = \langle si_t, BPS_t, C_t, I_t, P_t, H_t, R_t \rangle$ (notation $\mathbf{s \bigcup t}$) is the service $\boldsymbol{st} = \langle si_{st}, BPS_s \cup BPS_t, C_s \cup C_t, I_s \cup I_t, P_s \cup P_t, H_s \cup H_t, R_s \cup R_t \rangle$, where interface $\boldsymbol{si_{st}}$ contains combined operations from $\boldsymbol{si_s}$ and $\boldsymbol{si_t}$

- ***Intersection:*** The intersection of services $\boldsymbol{s} = \langle si_s, BPS_s, C_s, I_s, P_s, H_s, R_i \rangle$ and $\boldsymbol{t} = \langle si_t, BPS_t, C_t, I_t, P_t, H_t, R_t \rangle$ (notation $\mathbf{s \cap t}$) is the service $\boldsymbol{st} = \langle si_{st}, BPS_s \cap BPS_t, C_s \cap C_t, I_s \cap I_t, P_s \cap P_t, H_s \cap H_t, R_s \cap R_t \rangle$, where interface $\boldsymbol{si_{st}}$ contains only operations that can be supported by the elements originally belonging to services ***s*** and ***t***.

Furthermore, *empty*, *disjoint*, *composite* and *atomic* services can be defined as follows:

- ***Empty service:*** service $\mathbf{s = \langle \varnothing, \varnothing \rangle}$ (notation ∅) is the empty service

- ***Disjoint services:*** services **s** and **t** are said to be disjoint if $\mathbf{s \cap t} = \varnothing$

- ***Composite service:*** service **s** with $\mathbf{SOR\ (s) \cup OR\ (s) \neq (}$ is said to be a composite service

- Atomic service: service s with SOR (s) (OR (s) = (is said to be an atomic service

D. Different Types of Service-Oriented Systems

Having defined the general case for any SO system, without enforcing any constraints on the structure, the following definitions now specify constraints for specific types of service-oriented systems. These definitions reflect the conformance of a given system design to the fundamental characteristics of SO paradigm.

□ DEFINITION 6.A (Partitioned SO System (PARSOS))

A system that is entirely partitioned into services (i.e. there exist no implementation elements that do not belong to a service) is considered a *partitioned service-oriented system (PARSOS)*, and can be formally defined as:

$$\textbf{\textit{PARSOS}} = \langle SOS, SER \rangle$$

is a partitioned service-oriented system, only if SOS = <SI, BPS, C, I, P, H, R> is a service-oriented system as per Definition 5, and *SER* is a collection of services *ser* of *SOS* such that: $\forall bps \in BPS\ (\exists ser \in SER\ (bps \in BPS_{ser})) \wedge \forall c \in C\ (\exists ser \in SER\ (c \in C_{ser})) \wedge \forall i \in I\ (\exists ser \in SER\ (i \in I_{ser})) \wedge \forall p \in P\ (\exists ser \in SER\ (p \in P_{ser})) \wedge \forall h \in H\ (\exists ser \in SER\ (h \in H_{ser}))$

□ DEFINITION 6.B (Pure SO System (PURSOS))

A system that is both partitioned as per Definition 6.A, and in which every implementation element is part of *one and only one* service (i.e. all services in the system are disjoint as specified within Definition 5) is considered to be a *pure service-oriented system (PURSOS)*. The *'Academic Management System' (AMS_PURSOS)* shown in Fig. 2 is an example of such a system consisting of eleven fully independent services that communicate with each other strictly via service interfaces. Formally,

$$\textbf{\textit{PURSOS}} = \langle SOS, SER \rangle$$

is a pure service-oriented system, only if SOS = <SI, BPS, C, I, P, H, R> is a service oriented system as per Definition 5 and *SER* is a collection of services *ser* of *SOS* such that: $\forall bps \in BPS\ (\exists ser \in SER\ (bps \in BPS_{ser})) \wedge \forall c \in C\ (\exists ser \in SER\ (c \in C_{ser})) \wedge \forall i \in I\ (\exists ser \in SER\ (i \in I_{ser})) \wedge \forall p \in P\ (\exists ser \in SER\ (p \in P_{ser})) \wedge \forall h \in H\ (\exists ser \in SER\ (h \in H_{ser})) \wedge \forall ser_i, ser_j \in SER\ (ser_i \cap ser_j = \varnothing)$.

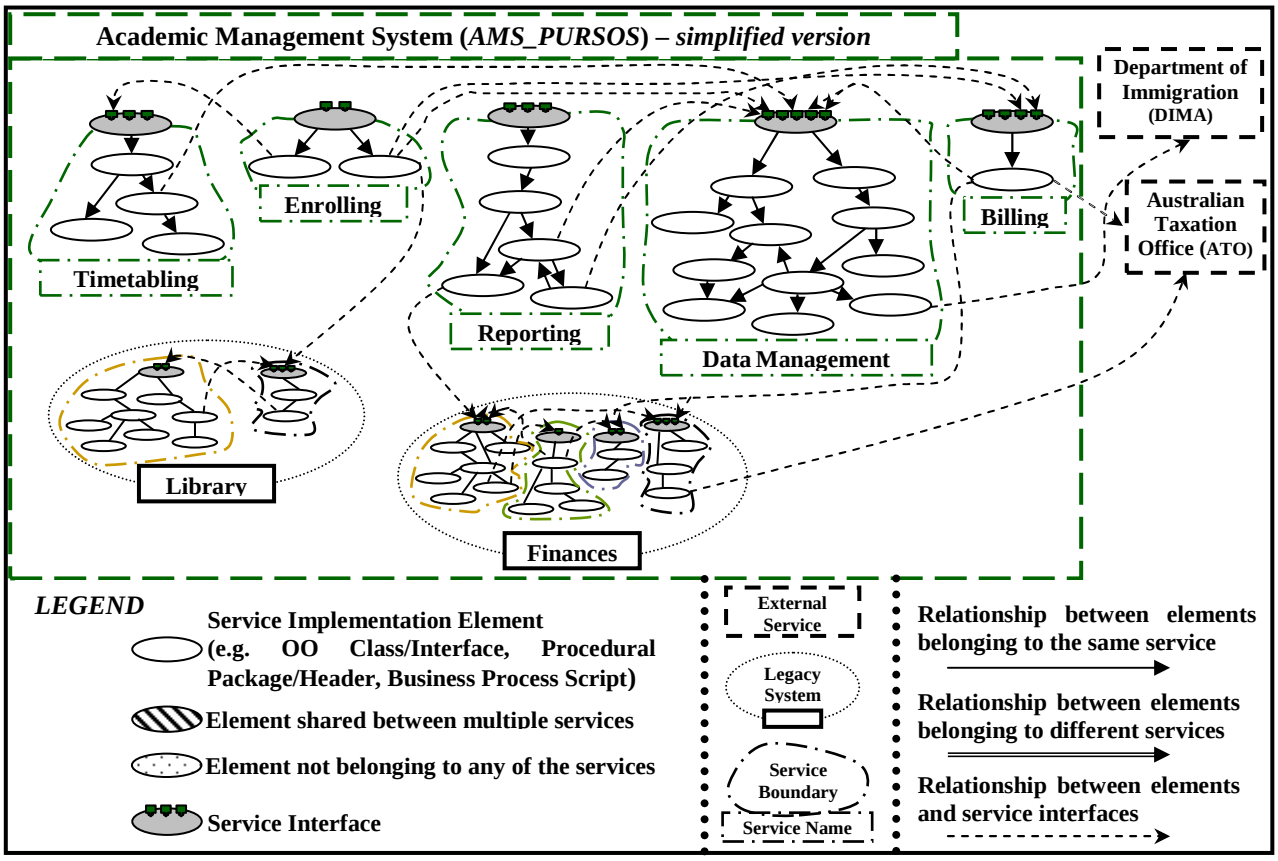

Figure 2. Example Pure Service-Oriented System Design

V. CUSTOMISING THE MODEL – BPEL4WS CASE

This section illustrates how the model presented in Section IV can be customised to support constraints imposed by a particular technology. Given the importance of Business Process Modelling (BPM) in the design process of service-oriented systems (refer to Section II), the paper shows the customisation for a specific business process language.

There are a large number of techniques and languages proposed for business process modelling ranging from workflow languages to UML and Petri Nets. These allow business processes to be designed and directly executed via middleware support. For example, in earlier work, Microsoft based XLANG on Pi-Calculus, IBM used Petri Nets with WSWL, and BPMI.org developed the Business Process Modelling Language (BPML) from scratch.

Subsequently, the Business Process Execution Language for Web Services (BPEL4WS) [4] was developed by a consortium of major software vendors. BPEL4WS combines ideas from XLANG and WSFL specifications and is arguably the most widely used business process modelling language. The newer version of BPEL, 'WS-BPEL 2.0' has been recently approved as the OASIS Standard [28].

Given the above, the following shows how the generic model can be modified for the case of BPEL4WS executable scripts, because such scripts affect the relationships between implementation elements with some relationships becoming obsolete and others being added to the model. More specifically:

i) every *bpel* script has a separate service interface which is the sole point of invocation. ii) *bpel* scripts cannot access implementation elements directly – a *bpel* script can only access/call service interfaces.

Based these constraints, the definition of the system and service structures (sub-section IV.A Definition 1), are refined, as are the sets of *common*, *possible*, and *impossible* relationships (sub-section IV.B Definition 4).

The rest of the definitions presented in Section IV remain the same since they are not influenced by BPEL specifics. The following shows the modified definitions with key differences highlighted in **bold**.

▫ DEFINITION 1 (BPEL) **BPEL4WS Case** (service structure)

Given a service-oriented system (*SYS*) as defined in Definition 1 (section IV.A), a *service* ***s*** can be defined as a sub-set of *SYS* consisting of just one service interface, and **either** one *bpel* script or collection of sets of classes (*C*); OO interfaces (*I*); procedural packages (*P*); and package interfaces/headers (*H*). Formally,

$$s = \langle si_s, \mathbf{bpel_s} \oplus (C_s, I_s, P_s, H_s) \rangle$$

is a service of SYS if and only if $si_s \in SI \wedge \mathbf{bpel_s \in BPEL} \wedge C_s \subseteq C \wedge I_s \subseteq I \wedge P_s \subseteq P \wedge H_s \subseteq H$.

▫ DEFINITION 4 (BPEL) **BPEL4WS Case** (relationships between different design artefacts in SO systems)

The following relationships can exist in service-oriented systems employing *BPEL4WS* scripts:

i) *Common* relationships $R_c = \langle CSI \cup SIC \cup CC \cup CI \cup IC \cup II \cup PSI \cup SIP \cup PP \cup PH \cup HH \cup \mathbf{BPELSI} \cup \mathbf{SIBPEL} \rangle$,

where CSI ⊆ C × SI, SIC ⊆ SI x C, CC ⊆ C × C, CI ⊆ C × I, IC ⊆ I × C, II ⊆ I × I, PSI ⊆ P × SI, SIP ⊆ SI × P, PP ⊆ P × P, PH ⊆ P × H, HH ⊆ H × H, **BPELSI ⊆ BPEL × SI, SIBPEL ⊆ SI × BPEL**

ii) *Possible* relationships R_p = <CP ⋃ PC ⋃ CH ⋃ PI>, *where* CP ⊆ C × P, PC ⊆ P x C, CH ⊆ C × H, PI ⊆ P × I

iii) *Impossible* relationships R_i = <SISI ⋃ SII ⋃ ISI ⋃ SIH ⋃ HSI ⋃ HP ⋃ HC ⋃ HI ⋃ IH ⋃ IP ⋃ **HBPEL ⋃ BPELH ⋃ CBPEL⋃ BPELC⋃ IBPEL⋃ BPELI⋃ PBPEL ⋃ BPELP**>, *where* SISI ⊆ SI × SI, SII ⊆ SI × I, ISI ⊆ I x SI, SIH ⊆ SI × H, HSI ⊆ H × SI, HP ⊆ H × P, HC ⊆ H × C, HI ⊆ H × I, IH ⊆ I × HI, IP ⊆ I × P , **HBPEL ⊆ H × BPEL, BPELH ⊆ BPEL × H ⊆ CBPEL ⊆ C × BPEL, BPELC ⊆ BPEL × C, IBPEL ⊆ I × BPEL, BPELI ⊆ BPEL × I, PBPEL ⊆ P × BPEL, BPELP ⊆ BPEL × P**

As was the case with the generic model, the set of overall coupling relationships **R** in a service-oriented design can therefore be represented as a union of all *common* and *possible* relationships: $\boldsymbol{R = R_c \cup R_p}$

VI. Model Application – Defining Metrics

In order to provide an example of applying the model described in Section IV, this section considers the derivation of two coupling and two cohesion metrics and their application to the example designs of Fig. 2-4. These metrics are part of an initial suite of service-oriented design metrics [34, 36], and are intended to predict the maintainability [21] of SO software.

Note that these metrics had to be defined since existing metrics are not directly applicable to the service-oriented designs due to the unique characteristics of SOC discussed in Section II. This was demonstrated in an exploratory empirical study [32] in which existing metrics [10, 26] were unable to sufficiently distinguish the structural properties of qualitatively different service-oriented designs.

A. Background

Lack of systematic guidelines and processes for the development of service-oriented systems can potentially result in decreased quality of produced software especially in terms of its *maintainability [21]*. This is significant given that software maintenance is the most resource intensive phase of the SDLC [41].

In the past, it was shown that design-level coupling and cohesion can have a causal impact on specific external quality attributes (such as maintainability) in both, the Procedural and OO paradigms [9, 15, 19]. Therefore, a suite of SO design-level *coupling* and *cohesion* metrics was derived in previous work [34, 36] in order to: i) predict maintainability of the final products in terms of analysability, changeability, stability, and testability [21]; ii) evaluate alternative design structures early in the SDLC. In addition to providing a *precise mechanism for evaluating and controlling software quality*, such metrics can also provide *initial development guidelines that should be incorporated in a service-oriented development methodology* as was described in sub-section III.B.

B. Example Coupling Metrics

The two example metrics described below are part of an initial suite of SO coupling metrics (17 metrics in total [36]), which was defined and validated using the model presented here. Note that in the context of SOC, coupling can be defined as a *measure of the extent to which interdependencies exist between services in a SO system.*

□ **Example Metric M1: *System Purity Factor (SPURF)***

SPURF is a system level metric which measures the extent to which a software system SOS is partitioned according to the definition of a *Pure Service-Oriented System (PURSOS).* In general, as the number of implementation elements belonging to more than one service increases, the *analysability* and *stability* of the system decreases, since changes to one element will influence more than one service. Formally,

$$SPURF\ (SOS) = 1 - |IS\ (SOS)| / |SER|$$

where IS is the set of all the *intersected services* in the system *SOS*, which can be expressed as: *IS (SOS)* = {ser1 | ser1 ∈ SER ∧ ∃ser2 ∈ SER(ser1 ∩ ser2 ≠ ∅)}; and *SER* is a set of all the services of *SOS* (described in section IV.C Definition 5). Values of SPURF range from *zero* to *one*, where *one* is the best possible value indicating that all the elements in the system belong to at most one service. For example, if designing a new system using a *top-down* strategy and following strict principles of service encapsulation and reuse it is easier (and desirable) to arrive at a design similar to that of Fig. 2 as reflected by the high value of *SPURF* (***AMS_PURSOS***) = 1.

However, in real life development, many projects may need to follow *bottom-up* or *meet-in-the-middle* strategies [5] to leverage existing resources or legacy systems which may not be able to be effectively refactored, thereby breaking principles of service encapsulation and reuse. Therefore, such a design may end up looking more like Fig. 3 as reflected by *SPURF* (***AMS***) = 0.1.

□ **Example Metric M2: *Weighted Extra-Service Incoming Coupling of Element (WESICE)***

WESICE for a given service implementation element *e* of a service *s* is the weighted count of the number of system elements $e_1 \ldots e_n$ not belonging to the same service that couple to this element. A high incoming coupling will negatively influence *changeability* since $e_1 \ldots e_n$ will be dependent upon the implementation characteristics of service *s*. As such, the reuse of the services containing external elements $e_1 \ldots e_n$ will be limited. Formally,

$$WESICE\ (e) = |\{(e, e1) * WeightFactor \mid \{e, e1\} \in IR(s)\}|$$

where IR(s) is the set of inter-service incoming direct relationships for service *s* (section IV.B Definition 4.C); and *WeightFactor* is a value assigned to different types of relationships based on their perceived influence on system coupling. For example, *CP* (OO Class → Procedural Package) type of relationship is weighted higher than *IC* (OO Interface → OO Class) type since the coupling is expected to be 'stronger' in the former case [36].

In contrast to SPURF, WESICE is an element level measure and as such is useful for identifying trouble spots or prioritising areas of the system that need additional investigation. For example, WESICE (***c1***) = 3 x 3 = 9 in

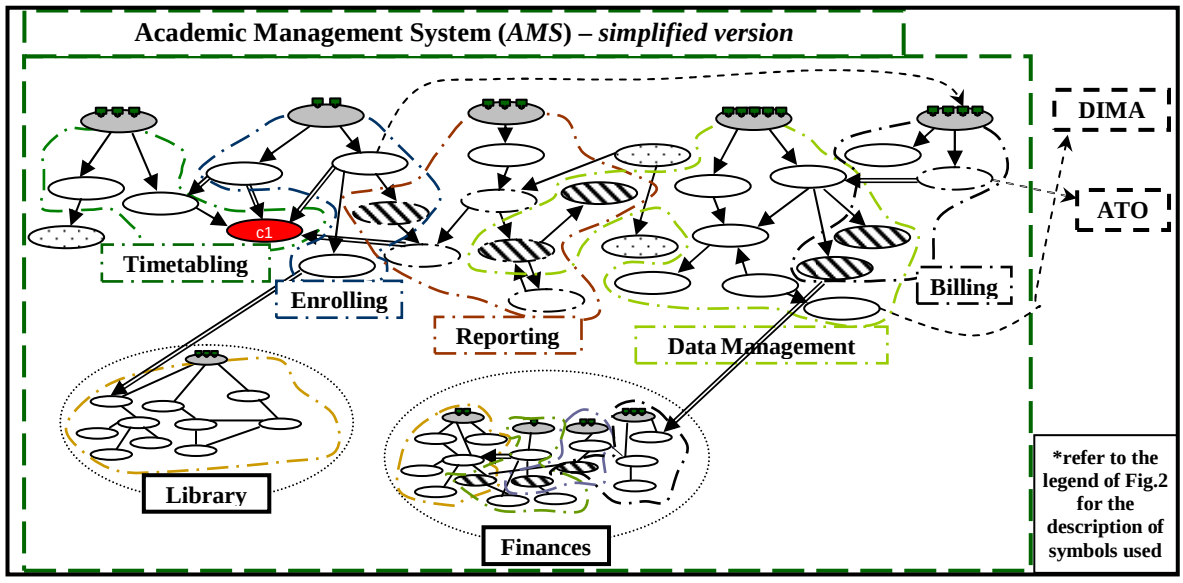

Figure 3. Example Non Pure Service-Oriented System Design

the design shown in Fig. 3 (where class $c1 \in C_{Timetabling}$ being accessed by 3 external elements), meaning that this is a potential design problem that must be addressed prior to implementation, or refactored during maintenance.

C. Example Cohesion Metrics

The two example metrics presented below are part of an initial suite of SO cohesion metrics (6 metrics in total [34]), which was again defined and validated using the proposed model, where cohesion can be defined as a *measure of the degree to which the operations exposed in a service interface conceptually belong together*.

Such metrics can be mapped to the qualitative categories of service-oriented cohesion proposed in [34] (Coincidental (weakest), Logical, Communicational, External, Sequential, Implementation, and Conceptual(strongest)), thereby providing a strong correlation between the metrics and our experienced-based understanding of the concept of service cohesion. The detailed explanation of the cohesion categories and all associated metrics is beyond the scope of this paper and can be found in [34], but generally the metrics values will range from 0 to 1, where value 1 indicates the strongest possible degree of cohesiveness for a particular cohesion category, and 0 indicates total lack of cohesion. For example, the following metrics should reflect the Implementation and External categories of cohesion.

- **Example Metric M3: *Strict Service Implementation Cohesion (SSIC)***

SSIC metric quantifies cohesion of a given service *(s)* based on the cohesiveness of the operations exposed in its interface *(si_s)*, as reflected by the associated implementation elements *($BPS_s \cup C_s \cup I_s \cup P_s \cup H_s$)*. A service is deemed to be highly cohesive when all service operations *($SO(si_s)$)* are implemented by the same implementation elements. Formally,

$$SSIC(s) = |IC(s)| / (|(BPS_s \cup C_s \cup I_s \cup P_s \cup H_s)| * |SO(si_s)|)$$

where $IC(s)$ is the set of all shared service implementation elements derived by intersecting all collaborations c of service s. Formally: $IC(s) = \{e \mid e \in (BPS_s \cup C_s \cup I_s \cup P_s \cup H_s) \wedge \forall\, c_{soA}, c_{soB} \in CO(si_s)\ (e \in c_{soA} \cap c_{soB})\}$. Refer to sub-section IV.B Definition 4.E for the description of c and $CO(si_s)$.

As an example, SSIC (***EnrollStudent***) = 11/(3*9) = 0.4 in Fig. 4 indicating an average cohesiveness of a service in respect to the Implementation category of cohesion (which is reflected by service interface operations being implemented by the same implementation elements [34], where shared/common elements are represented by black ovals).

- **Example Metric M4: *Service Interface Usage Cohesion (SIUC)***

SIUC metric quantifies cohesion of a given service *(s)* based on the cohesiveness of the operations exposed in its interface *(si)*, as reflected by the behavioural communication (usage) pattern of service consumers. A service is deemed to be highly cohesive when all service operations *(SO (si))* are invoked by every client (service consumer). Clients will most likely be either internal or external services, but in theory, any piece of executable software can be considered as the client if it invokes operations of services being measured. Formally,

$$SIUC(s) = INV(clients, SO(si_s)) / (num\_clients * |SO(si_s)|)$$

where $SO(si_s)$ is the set of all service operations as defined in sub-section IV.A Definition 2; *INV(clients, SO(sis))* is the function which computes the sum of all used operations calculated on per client basis; *num_clients* is the total number of clients invoking operations of *s*. For example, SIUC (***EnrollStudent***) = 3/3 = 1 in Fig. 4 indicating a strongest possible degree of External cohesiveness (which is reflected by service operations being used by the same consumers [34]).

VII. Related Work

A widely-referenced and well-documented model of software design was proposed by Briand et al. [8]. In this model any system *S* can be captured as a pair <*E*, *R*>,

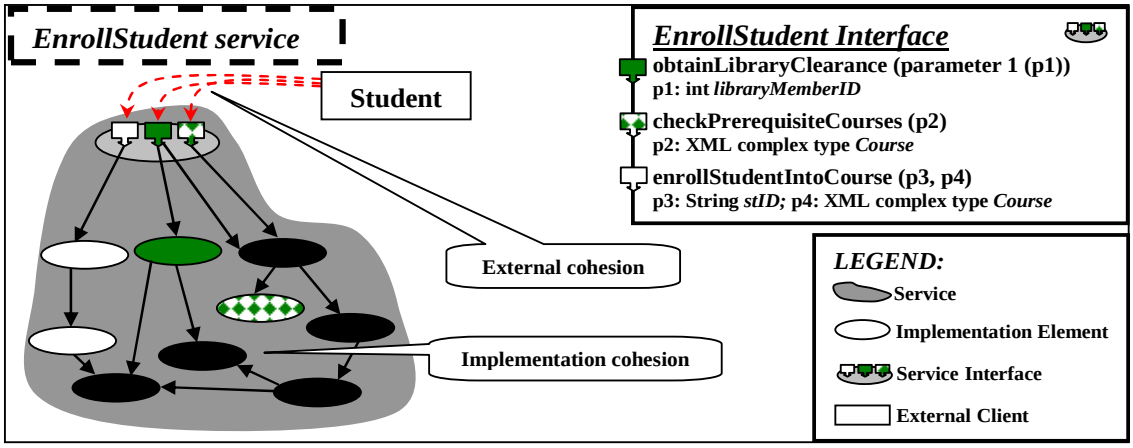

Figure 4. Example Categories of Service Cohesion

where E represents the set of elements of S, and R is a binary relation on E ($R \subseteq E \times E$) representing the relationships between the elements of S. Briand et al. used this model to assist in the definition of structural properties of software design, and the derivation of associated measures [8, 9]. Note however, that this model was a generic representation of a software system and therefore intended to be extended for a particular paradigm (as was done in this work).

For example, in previous work, the model was extended by Rossi and Fernandez [37] to represent a generalised distributed system structure. Also, Morasca used Briand's model when deriving measures for concurrent systems [27]. However, these models treat applications as a collection of local or remote components independent of specific implementation architecture, which makes them inapplicable to service-oriented systems. This is because services are typically implemented using a range of different technologies and paradigms, therefore service implementation elements should be treated as separate units based on their underlying paradigm rather than being combined into one single generic element as was done in [2, 8, 27, 37].

Additionally, many other formalisms can be used to model SOA-based systems. For example, there are a number of approaches that propose formal logical models for capturing semantics of service interfaces (e.g. OWL-S [25]) in order to assist in dynamic discovery, binding, and orchestration of services. Also, there are a number of models that cover communicational and collaborative aspects of services using formal representations based on Petri-Nets or finite state automata [7]. Furthermore, some recent approaches model web services using software architecture notations with formal behavioural contract relationships between service components [24]. These models are not concerned with the design and implementation of individual services, instead treating them as "black boxes" or nodes in a workflow. As such, these models do not capture the structural characteristics of SO designs.

In contrast, the model proposed by Briand et al. was intended to capture the structure of software designs, therefore the decision was made to extend Briand's model since this model: i) was successfully used before; ii) allows application of all set-theoretic operations; iii) maps better to the practical design and implementation aspects of SO development compared to the abstract non-formal, architectural representations of SO systems, such as the one described by Arsanjani [5].

VIII. Conclusions and Future Work

This paper has described a formal model covering design artefacts and associated relationships in service-oriented systems. The model formalises concepts of SO design according to the fundamental principles of service-orientation presented in the paper, thus promoting a better understanding of the service-oriented paradigm. More importantly, the model supports the precise mathematical definition and theoretical validation of metrics for quantifying the structural properties of service-oriented designs. The metrics lay the foundation for a systematic service-oriented software design methodology, and also provide means for evaluating such a methodology once it is derived. A number of steps required to define such a methodology were presented in this paper.

In order to demonstrate how the model can be used when defining metrics, four example metrics from an initial suite of service-oriented coupling and cohesion design metrics were presented. Also, the model aims to be generic whilst providing scope to capture the impact of specific technologies. One such example based on BPEL4WS was presented in this paper and other variations such as component based technologies like Enterprise Java Beans (EJB) could be added in the future with little change to the presented default model.

In future work, the authors plan to further evaluate the completeness of the proposed model by using it to represent design structures of large-scale commercial service-oriented systems. Such representations will than be measured using metrics in order to identify important design characteristics influencing the quality of service-oriented software products. Furthermore, the model will be augmented with dynamic aspects of SOA in order to capture the dynamic behaviour of service-oriented systems when parameterised by usage and physical deployment context. This will allow derivation of the predictive models for estimating efficiency and reliability of service-oriented software.

APPENDIX A NOTATION USED IN THE PAPER

Set Theory

A = {a, b, …, z} - set; **a ∈ A** - set membership
|A| - set cardinality; **A ⊆ B** - A is a subset of B
A ⊂ B - A is a proper subset of B (A not equal to B)
A = ∅ - empty set; **A ⋃ B** - union
A ∩ B - intersection; **A ∩ B = ∅** - disjoint set
A x B - Cartesian product; **A - B** - relative complement

Propositional Calculus

∧ - AND (conjunction); **∨** - OR (inclusive disjunction)
⊕ - exclusive OR (exclusive disjunction)

Predicate Calculus (first-order)

∀ - (universal quantification – "for all")
∃ - (existential quantification – "there exists")

Miscellaneous

<…> symbol has been used to represent elements of a set in a case when these elements are sets themselves, e.g. SOS = <SI; …, R>.

{…} symbol has been used to represent atomic elements of a set, e.g. SI = {si_1, …, si_n).

(…) symbol has been used to indicate an ordered pair of elements (representing an edge on the design graph, i.e. a relationship between two implementation elements), e.g. R = {(a,b), …, (y, z)}.

⟨⟩ symbol has been used to represents *service membership* (refer to sub-section IV.A Definition 1). e.g. a ⟨⟩ s).

ACKNOWLEDGMENT

This project is funded by the ARC (Australian Research Council), under Linkage scheme no. LP0455234.

REFERENCES

[1] M. Acharya, et al., "SOA in the Real World – Experiences", presented at 4th Intl. Conference on Service Oriented Computing (ICSOC05), Chicago, USA, 2005.

[2] B. E. Allen, "Measuring Graph Abstractions of Software: An Information-Theory Approach", presented at 8th IEEE Symposium on Software Metrics, Ottawa, Canada, 2002.

[3] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services: Concepts, Architectures and Applications*. Heidelberg, Germany: Springer-Verlag, 2004.

[4] T. Andrews, et al., "Business Process Execution Language for Web Services, Version 1.1." BEA Systems, IBM Corp., Microsoft Corp., SAP, Siebel Systems, 2003. http://www-128.ibm.com/developerworks/library/specification/ws-bpel/.

[5] A. Arsanjani, "Service-oriented modeling and architecture: how to identify, specify, and realize services for your SOA", IBM whitepaper, 2004. ftp://www6.software.ibm.com/software/developer/library/ws-soa-design1.pdf.

[6] D. K. Barry, *Web services and service-oriented architectures: the savvy manager's guide*. San Francisco, CA: Morgan Kaufmann; Elsevier Science, 2003.

[7] D. Berardi, F. De Rosa, L. De Santis, and M. Mecella, "Finite State Automata As Conceptual Model For E-Services", *Journal of Integrated Design and Process Science*, IOS Press, vol. 8 (2/2004), pp. 105-121, 2004.

[8] L. C. Briand, S. Morasca, and V. R. Basili, "Property-Based Software Engineering Measurement", *IEEE Transactions on Software Engineering*, vol. 22 (1), pp. 68-86, 1996.

[9] L. C. Briand, J. Wust, J. Daly, and V. Porter, "Exploring the relationship between design measures and software quality in object-oriented systems", *Journal of Systems and Software*, Elsevier Science, vol. 51 (3), pp.245-273, 2000.

[10] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object-Oriented Design", *IEEE Transactions on Software Engineering*, vol. 20 (6), pp. 476-493, 1994.

[11] J. Eder, G. Kappel, and M. Schrefl, "Coupling and cohesion in object-oriented systems", presented at ACM Conference on Information and Knowledge Management (CIKM), Baltimore, USA, 1992.

[12] M. Endrei, et al., "Patterns: Service-Oriented Architecture and Web Services", IBM Redbooks, 2004. http://www.redbooks.ibm.com/redbooks/SG246303/.

[13] T. Erl, *Service-Oriented Architecture: A field guide to integrating XML and Web services*. Upper Saddle River, NJ: Prentice Hall PTR, 2004.

[14] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*. Indiana, USA: Prentice Hall PTR, 2005.

[15] F. Fioravanti and P. Nesi, "Estimation and prediction metrics for adaptive maintenance effort of object-oriented systems", *IEEE Transactions on Software Engineering*, vol. 27 (12), pp. 1062-1084, 2001.

[16] L. R. Foulds, *Graph Theory Applications*. New York: Springer-Verlag New York, Inc., 1992.

[17] M. Fowler, D. Rice, and D. Hoang, *Patterns of enterprise application architecture*. Boston: Addison-Wesley, 2003.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Reading, Mass.: Addison-Wesley, 1995.

[19] M. Genero, J. Olivas, M. Piattini, and F. Romero, "Using Metrics to Predict OO Information Systems Maintainability", presented at 13th International Conference on Advanced Information Systems Engineering (CAiSE 2001), Interlaken, Switzerland, 2001.

[20] R. Heffner, "Planned SOA Usage Grows Faster Than Actual SOA Usage", Forrester Research, 2007. http://www.forrester.com/Research/Document/Excerpt/0,7211,41686,00.html.

[21] ISO/IEC, "ISO/IEC 9126-1:2001 Software Engineering: Product quality - Quality model", International Standards Organisation, Geneva, 2001.

[22] I. H. Kruger and R. Mathew, "Systematic development and exploration of service-oriented software architectures", presented at 4th Working IEEE/IFIP Conference on Software Architecture, Oslo, Norway, 2004.

[23] M. Lehmann, "Deploying large-scale interoperable Web Services infrastructures", *Web Services Journal*, vol. 5 (1), pp. 10-15, 2005.

[24] S. Ling, I. Poernomo, and H. Schmidt, "Describing Web Services Architectures through Design-by-Contract", presented at 18th Intl. Symposium On Computer and Information Sciences (ISCIS'03), Antalya, Turkey, 2003.

[25] D. B. Martin, et al., "OWL-S: Semantic Markup for Web Services", World Wide Web Consortium (W3C), 2004. http://www.w3.org/TR/owl-features/.

[26] T. J. McCabe and A. H. Watson, "Software Complexity", *Journal of Defense Software Engineering*, vol. 7 (12), pp. 5-9, 1994.

[27] S. Morasca, "Measuring Attributes of Concurrent Software Specifications in Petri Nets", presented at 6th International Symposium on Software Metrics, Boca Raton, USA, 1999.

[28] OASIS, "Web Services Business Process Execution Language Version 2.0", Organization for the Advancement of Structured Information Standards, 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[29] L. Padgham and M. Perepletchikov, "Prioritisation mechanisms to support incremental development of agent systems", *International Journal of Agent-Oriented Software Engineering*, Inderscience, vol. 1 (3), 2007.

[30] M. P. Papazoglou and A. D. Georgakopoulos, "Service-Oriented Computing", *Communications of the ACM*, vol. 46 (10), pp. 24-28, 2003.
[31] J. Pasley, "How BPEL and SOA are changing Web services development", *Internet Computing, IEEE*, vol. 9 (3), pp. 60-67, 2005.
[32] M. Perepletchikov, C. Ryan, and K. Frampton, "Comparing the Impact of Service-Oriented and Object-Oriented Paradigms on the Structural Properties of Software", presented at 2nd International Workshop on Modeling Inter-Organizational Systems (MIOS'05), Ayia Napa, Cyprus, 2005.
[33] M. Perepletchikov, C. Ryan, and Z. Tari, "The Impact of Software Development Strategies on Project and Structural Software Attributes in SOA." presented at 2nd INTEROP Network of Excellence Dissemination Workshop (INTEROP'05), Ayia Napa, Cyprus, 2005.
[34] M. Perepletchikov, C. Ryan, and K. Frampton, "Cohesion Metrics for Predicting Maintainability of Service-Oriented Software", presented at 7th International Conference on Quality Software (QSIC2007), Portland, USA, 2007.
[35] M. Perepletchikov, C. Ryan, K. Frampton, and H. Schmidt, "A Formal Model of Service-Oriented Design Structure", presented at 18th Australian Conference on Software Engineering (ASWEC 2007), Melbourne, Australia, 2007.
[36] M. Perepletchikov, C. Ryan, K. Frampton, and Z. Tari, "Coupling Metrics for Predicting Maintainability in Service-Oriented Designs", presented at 18th Australian Conference on Software Engineering (ASWEC 2007), Melbourne, Australia, 2007.
[37] P. Rossi and G. Fernandez, "Definition and Validation of Design Metrics for Distributed Applications", presented at 9th International Software Metrics Symposium, Sydney, Australia, 2003.
[38] J. Rumbaugh, et al., *Object Oriented Modelling and Design*. NJ: Prentice-Hall, 1991.
[39] H. Schmidt, "Trustworthy components: compositionality and prediction", *Journal of Systems and Software*, Elsevier Science Inc., vol. 65 (3), pp. 215-225, 2003.
[40] M. P. Singh and M. N. Huhns, *Service-Oriented Computing: Semantics, Processes, Agents*. West Sussex, England: John Wiley & Sons, 2005.
[41] H. van Vliet, *Software Engineering:Principles and Practices*, 2nd Edition ed. West Sussex, England: John Wiley & Sons, 2000.
[42] O. Zimmermann, P. Krogdahl, and C. Gee, "Elements of Service-Oriented Analysis and Design: an interdisciplinary modeling approach for SOA projects", IBM - whitepaper, 2004. http://www-128.ibm.com/developerworks/library/ws-soad1/.

Mikhail Perepletchikov received B.App.Sc. (Computer Science) Honours 1st Class degree from RMIT University, Melbourne, Australia in 2004. Upon completion of his honours degree, Mikhail had worked as a casual Software Engineer for the Continuum TAI consulting firm in Melbourne, Australia. He is currently a full-time PhD candidate in the School of Computer Science and Information Technology, RMIT University working in the area of software metrics for service-oriented applications. Mikhail's research interests include Agent-Oriented and Service-Oriented software development methodologies and processes, requirements engineering, and metrics for predicting software quality early in the SDLC.

Mr Perepletchikov is a member of the Australian Computer Society (ACS), the Institute of Electrical and Electronics Engineers (IEEE), and the IEEE Computer Society. He is a current recipient of the Australian Postgraduate Award (Industry) scholarship awarded to outstanding PhD candidates.

Caspar Ryan received his PhD in Computer Science from RMIT University, Melbourne, Australia in 2002 and is now a Senior Lecturer at the same institution. His PhD involved the study of expert and novice OO designers which lead to his current software engineering research in metrics and methodology for SOA. Caspar also has significant practical experience in distributed systems and mobile computing, having managed a project in the Australian Telecommunications CRC from 2004-2006. He was an inventor on three provisional U.S and Australian patents filed in relation to this project, the ideas of which were embodied in an operational prototype demonstrated at the CeBIT Australia exhibition in Sydney in 2005. Caspar has been an invited speaker at various universities and has also presented to industry audiences at events such as the Enterprise Java Australia seminars.

Dr Ryan is a member of the Australian Computer Society (ACS) and the Association of Computing Machinery (ACM).

Keith Frampton earned his B.Sc. in Computer Science in 1980 from Monash University in Melbourne, Australia. He has worked for over 25 years in different software engineering roles within Australia and throughout the world. For the last 15 years, he has worked with major domestic and international companies as an IT Architect and strategist and currently works for The Marlo Group, a specialist integration consulting group in Melbourne, Australia. Keith also worked for RMIT University part-time and was responsible for the content of their Masters of Enterprise Architecture, and is currently undertaking a PhD that explores distinguishing capabilities of better IT Architects. His other research interests include industrial usage of development methodologies, what skills do companies require, and the effective teaching of software engineering.

Mr Frampton is a member of the Australian Computer Society (ACS), the Association of Computing Machinery (ACM), Association of Information Systems, and the Enterprise Architects Association.

Heinz Schmidt received his PhD in Computer Science from Bremen University, Germany in 1988. He was a Research Fellow at the ICSI at UC Berkeley, a researcher at the German National Research Centre for Computer Science and the Australian CSIRO. He has been Professor of Software Engineering for over 12 years, at Monash University, Melbourne and now RMIT University, Melbourne, Australia. Heinz recognised in the field of Software Engineering (SE) for concurrent, parallel, and distributed, component-based systems. He has published over 120 articles and papers. He also has contributed to establishing the CBSE symposium series. Heinz is an editor of the IEEE Transactions on Software Engineering, Springer Knowledge and Information, and SDPS Journal of Integrated Design and Process Science. His interests include formal models, practical methods and advanced tools for large-scale systems and software architecture analysis and design, in particular extra-functional aspects of distributed parallel and real-time component-based software.

Prof. Schmidt is a member of the Institute of Electrical and Electronics Engineers (IEEE), the Association of Computing Machinery (ACM), and the German Engineering Society (VDI).