A Formal Model of Service-Oriented Design Structure

Mikhail Perepletchikov, Caspar Ryan, Keith Frampton, and Heinz W Schmidt
RMIT University
School of Computer Science and Informational Technology
Melbourne, Australia
{mikhailp, caspar, keithf, hws}@cs.rmit.edu.au

Abstract

Service-Oriented Computing (SOC) is a promising paradigm for developing enterprise software systems. The initial concepts of service-orientation have been described in the research and industry literature and software tools for assisting in the development of Service-Oriented (SO) applications are becoming more widely used. Nonetheless, a precise description of what constitutes a SO system is yet to be formally defined, and the design principles of SOC are not well understood. Therefore, this paper proposes a formal mathematical model covering design artefacts in service-oriented systems and their structural and behavioural properties. This model promotes a better understanding of SO concepts, and in particular, enables the definition of structural software metrics in an unambiguous, formal manner. Finally, although the proposed model is generic, it can be customised to support particular technologies as shown in this paper where the model was tailored for BPEL4WS implementation.

1. Introduction

Service-Oriented Computing (SOC) is emerging as a promising new software development paradigm [11, 26]. SOC is based on encapsulating application logic within independent, loosely coupled, stateless *services* that interact via messages using standard communication protocols, which can be orchestrated using business process languages [4, 19]. 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 typically more autonomous and hence decoupled from other services [11]. Furthermore, implementation inheritance and its complications, in particular in relation to compositionality [25] is not an issue, while it is in many component models.

Service-Oriented systems, in conjunction with supporting middleware, represent *Service-Oriented Architecture (SOA)*, a more abstract concept concerning how software services should be composed and orchestrated to fulfil a specific domain or business requirement. In SOA, instead of thinking of services as interfaces to software functionality that connect to other services, enterprises should consider services as enablers of *business processes* that reflect workflows within and between organisations.

One of the main advantages of SOA is its business alignment [5, 20]. 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. As such, they 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.

Nevertheless, despite these potential advantages, the design principles of SOC are yet to be well understood, with contradicting definitions and guidelines making it hard for software engineers and developers to work effectively with service-oriented concepts [4, 10, 11, 19]. Consequently, service-oriented systems are often developed in an ad-hoc fashion [12, 16, 27] resulting in low-quality software being released.

In contrast, established development paradigms, in particular Object-Orientation (OO), are more widely understood. This understanding is underpinned by formal models of OO design, which have facilitated the derivation of metrics for measuring structural attributes such as coupling and cohesion [2, 7, 9], which in turn

Proceedings of the 2007 Australian Software Engineering Conference (ASWEC'07)
0-7695-2778-7/07 $20.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

has facilitated the measurement and prediction of software quality [8]. However, such models and metrics are not immediately applicable to SOC since the design of Service-Oriented systems is conceptually different to OO, as described in section 2.

Therefore, the main contribution of this paper is the proposition of a formal model covering *structural* and *behavioural* properties of the *design artefacts in service-oriented systems*. This model extends the generic model proposed by Briand et al. [7], where a software design is represented as a relational system, consisting of elements, relations and binary operations. The extensions are based on the core characteristics of service-orientation as described by existing literature [10, 11, 19, 26] and the authors' experience with developing SOA-based applications [21].

There are two main benefits of the proposed model. Firstly, it formalises the structural design concepts surrounding service-oriented development, which, being an emerging paradigm, have not always been consistently expressed nor well understood [1, 12, 16, 22]. Secondly, the model allows software metrics related to the structural properties of SO design to be defined in a precise mathematical manner.

Finally, although the proposed model was designed to be as generic and technology agnostic as possible, it can be readily customised to support specific technologies, as shown in Section 4 where the model was tailored for the specific case of BPEL4WS.

The remainder of this paper is organised as follows. Section 2 provides a brief overview of work on formalising software design thus providing a rationale for this research. Section 3 presents a generic model of service-oriented design, which is then specialised in section 4 for the specific case of BPEL4WS. Section 5 shows two example metrics for measuring structural coupling of SO designs that have been defined based on the proposed model. Finally, Section 6 closes with conclusions and a discussion of future work.

2. Related work

According to a generic model proposed by Briand et al. [7] any software system *S* can be represented as a pair <E, R>, 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 used this model to assist in the definition of structural properties of software design, and the derivation of associated measures. Note however, that this model was a generic representation of a software system and therefore intended to be extended for a particular paradigm.

For example, in previous work, the model was extended by Rossi and Fernandez [24] to represent a generalised *distributed* system structure. Also, Morasca [18] used Briand's model when deriving measures for *concurrent* systems. However, neither the original nor the extended models are immediately applicable to SOC because they treat applications as a collection of local or remote components independent of specific implementation architecture.

In contrast, services can be implemented using a range of different technologies and development paradigms, which 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 [9]. This is 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. As such, the decision was made to treat each service implementation element as a separate unit rather than combining all artefacts into one single generic element as was done in [2, 7, 18, 24]. Furthermore, business process scripts were included as a separate implementation element type since they were expected to have characteristics that differentiate them from procedural or OO implementation elements. Consequently, the concept of a *service implementation element* or module was sub-divided into: *business process scripts, OO classes,* and *procedural packages* (collection of procedures).

Also, SOC has two additional unique characteristics that should be reflected in the model of a SO system:

Firstly, SO Computing has more levels of abstraction than previous development paradigms. For example, the procedural paradigm has only one main level of abstraction: the *procedure* (element). The OO paradigm has two levels of abstraction: *methods* (elements) which are aggregated into *classes* (modules). In contrast, SOC introduces a third level of abstraction and encapsulation: the service, in which *operations* (methods) are aggregated into *elements* (classes/procedures, etc.) that implement the functionality of a *service* exposed through its *interface*.

Secondly, it is clear that an *interface* should be a first class design artefact in service-oriented systems which are based on loosely coupled services. For example one of the primary motivations for 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

IEEE COMPUTER SOCIETY

are commonly used to implement services), coupling through interface types again has a lower coupling than through class types, as widely advocated by experienced practitioners [14]. Thus, in addition to *service interfaces*, the *OO interfaces* and *Procedural headers* should be treated as separate entities.

Finally, note that many other formal models can be used to represent services in Service-Oriented Architectures. For example, there are a number of approaches that propose formal logical models for capturing semantics of service interfaces (e.g. OWL-S (http://www.daml.org/services/owl-s/1.0/)) in order to assist in dynamic discovery, binding, and orchestration of services. Furthermore, there are a number of models that cover communicational and collaborative aspects of services using formal representations based on Petri-Nets [17] or finite state automata [6]. Such 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 design. In contrast, the model presented in this paper is intended to capture the *structure of SO designs*, but does not cover other aspects of SOA such as negotiation, dynamic discovery and composition, and semantic interfaces since these non-functional aspects are not explicitly represented in SO design structure.

As a result, in this work, the decision was made to extend Briand's model since this model: i) is simple and intuitive; ii) was successfully used before; iii) allows application of key set-theoretic operations; iv) maps better to the practical design and implementation aspects of SO development compared to the abstract non-formal, architectural representations of SO system design, such as the one described by Arsanjani [4].

3. Formal model of SO design

Given the rationale above, this section extends the generic model of Briand [7] to capture the design structure of a *service-oriented system* as a bi-directional graph expressed using set-theoretic notation. Vertices in the graph symbolise design artefacts 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, representing both structural and behavioural dependencies. Finally, from the graph of a *service-oriented system*, specific sub-graphs representing individual *services* are derived.

Figure 1 shows an example graph representing the arbitrary design structure of a service-oriented system with two marked sub-graphs representing its

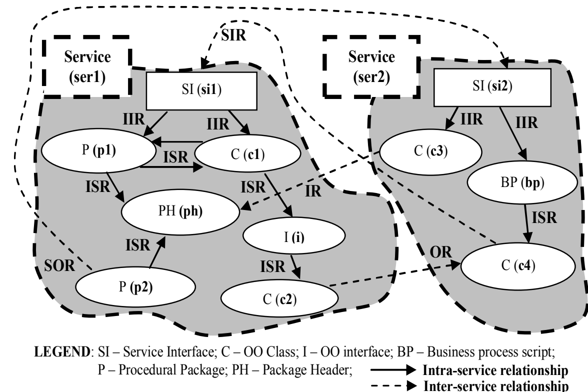

Figure 1. Structure of Service-Oriented System (SOS)

constituent services. As described above, this graph can be expressed using conventional graph theory [13] where each graph is 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, the service-oriented design structure of the *SOS* graph illustrated in Figure 1 consists of a vertex set V(*SOS*) = {si1, si2, p1, c1 c3, ph, i, bp, p2, c2, c4}, and an edge set E(*SOS*) = {(si1, p1), (si1, c1), (c1, p1), (p1, c1), (c1, i), (i, c2), (p2, ph), (p2, si2), (c3, ph), (si2, c3), (si2, bp), (bp, c4), (c4, si1)}.

According to graph theory, a sub-graph *sg* of a graph *G* is a graph whose vertex and edge sets are subsets of those of *G*. Figure 1 shows two such sub-graphs of *SOS*, which represent services *ser1* and *ser2*. For example, *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), (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/class which is physically encapsulated within an OO class/package. As such, 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; flow charts or data flow diagrams; or by tracing the actual executable code where available.

As an example consider again the sample design of Figure 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*

IEEE COMPUTER SOCIETY

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 largely 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.

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. However the concept of a *component* implementation element could easily be added as a first class element if necessary. The same applies to other popular implementation technologies such as scripting languages, since they can readily be classified as OO or procedural implementations.

Finally, the model is presented in stages to improve readability, with section 3.1 describing the entities representing design artefacts; section 3.2 the relationships between them; and section 3.3 providing a complete model that combines the concepts from the previous two subsections.

3.1. 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 are subsets of the sets comprising the total elements of the system, with the exception of the service interface which is a single element because a service logically has only a single service interface. Formally,

− a *system* ***SYS* = <SI, BPS, C, I, P, H>**, *where*
SI = the set of all service interfaces in the system, BPS = the set of all business process scripts, C = the set of all OO classes, I = the set of all OO interfaces, P = the set of all Procedural packages, H = the set of all Package headers.

− given a system (*SYS*), a *service* $\mathbf{s = \langle si_s, BPS_s, C_s, I_s, P_s, H_s \rangle}$ *is a service of SYS if and only if* $si_s \in SI \wedge (BPS_s \subseteq BPS \wedge C_s \subseteq C \wedge I_s \subseteq I \wedge P_s \subseteq P \wedge H_s \subseteq H) \wedge (BPS_s \cup C_s \cup I_s \cup P_s \cup H_s \langle\rangle s)$.

Note that $\langle\rangle$ 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 Definition 5) of $\mathbf{si_s}$ i.e. if ***e*** belongs to some collaboration sequence $\mathbf{cs \in CS}$ as part of collaboration $\mathbf{c = \langle Param(so \in SO(si_s)), CS \rangle}$ (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 Figure 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 $\mathbf{e \in SI \cup BPS \cup C \cup I \cup P \cup H}$ *let* ***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* $\mathbf{si \in SI}$ *let* ***SO(si)*** *be the set of service operations* ***so*** *of service interface* ***si***.

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* $\mathbf{o \in O(e)}$ *let* ***Param(o)*** *be the set of generic parameters of* ***o***.

In addition, parameters can be defined for different operation types, e.g. *For each service operation* $\mathbf{so \in SO(si)}$ *let* ***Param(so)*** *be the set of parameters of* ***so***.

3.2. 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 shown in Definition 5.

Proceedings of the 2007 Australian Software Engineering Conference (ASWEC'07)
0-7695-2778-7/07 $20.00 © 2007 IEEE
IEEE COMPUTER SOCIETY

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, the relationships are described as follows:

Firstly, a set of *common* relationships $\mathbf{R_c}$ represents relationships that are likely 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 (CC) or through an interface which is implemented by a specific class (CI). Formally,

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

$\mathbf{CSI} \subseteq C \times SI$, $\mathbf{SIC} \subseteq SI \times C$, $\mathbf{CC} \subseteq C \times C$, $\mathbf{CI} \subseteq C \times I$, $\mathbf{IC} \subseteq I \times C$, $\mathbf{II} \subseteq I \times I$, $\mathbf{PSI} \subseteq P \times SI$, $\mathbf{SIP} \subseteq SI \times P$, $\mathbf{PP} \subseteq P \times P$, $\mathbf{PH} \subseteq P \times H$, $\mathbf{HH} \subseteq H \times H$, $\mathbf{BPSSI} \subseteq BPS \times SI$, $\mathbf{SIBPS} \subseteq SI \times BPS$, $\mathbf{BPSBPS} \subseteq BPS \times BPS$

Note that × symbol represents a Cartesian product between two given sets. 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 Figure 1, where each single relationship is represented as the *ordered pair (source, destination)*.

Secondly, a set of *possible* relationships $\mathbf{R_p}$ is specified which comprises 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,

– $\mathbf{R_p} = \langle \mathbf{CP} \cup \mathbf{PC} \cup \mathbf{CH} \cup \mathbf{CBPS} \cup \mathbf{BPSC} \cup \mathbf{BPSI} \cup \mathbf{PBPS} \cup \mathbf{BPSP} \cup \mathbf{BPSH} \cup \mathbf{PI} \rangle$, *where*

$\mathbf{CP} \subseteq C \times P$, $\mathbf{PC} \subseteq P \times C$, $\mathbf{CH} \subseteq C \times H$, $\mathbf{CBPS} \subseteq C \times BPS$, $\mathbf{BPSC} \subseteq BPS \times C$, $\mathbf{BPSI} \subseteq BPS \times I$, $\mathbf{PBPS} \subseteq P \times BPS$, $\mathbf{BPSP} \subseteq BPS \times P$, $\mathbf{BPSH} \subseteq BPS \times H$, $\mathbf{PI} \subseteq P \times I$

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) cannot call another service interface (SI) (or other explicit interface types such as OO interface (I) or Procedural header (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 implementation elements. Finally, it is impossible to have a relationship from an OO interface (I) to the elements belonging to different development paradigms such as Procedural packages (P) and headers (H), and Business Process Scripts (BPS). For completeness these are listed below.

– $\mathbf{R_i} = \langle \mathbf{SISI} \cup \mathbf{SII} \cup \mathbf{ISI} \cup \mathbf{SIH} \cup \mathbf{HSI} \cup \mathbf{HP} \cup \mathbf{HC} \cup \mathbf{HI} \cup \mathbf{HBPS} \cup \mathbf{IH} \cup \mathbf{IP} \cup \mathbf{IBPS} \rangle$, *where*

$\mathbf{SISI} \subseteq SI \times SI$, $\mathbf{SII} \subseteq SI \times I$, $\mathbf{ISI} \subseteq I \times SI$, $\mathbf{SIH} \subseteq SI \times H$, $\mathbf{HSI} \subseteq H \times SI$, $\mathbf{HP} \subseteq H \times P$, $\mathbf{HC} \subseteq H \times C$, $\mathbf{HI} \subseteq H \times I$, $\mathbf{HBPS} \subseteq H \times BPS$, $\mathbf{IH} \subseteq I \times HI$, $\mathbf{IP} \subseteq I \times P$, $\mathbf{IBPS} \subseteq I \times BPS$

Note that these sets of relationships are based on current practice and the experience of the authors, but are not considered definitive and could change in response to changing technology.

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 between system elements $\mathbf{R} = \mathbf{R_c} \cup \mathbf{R_p}$. Accordingly, the relationships belonging to a particular service *s* can be represented as: $\mathbf{R_s} = \mathbf{R_{cs}} \cup \mathbf{R_{ps}}$

The following definitions address relationships involving one or more services.

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

An example of IIR relationships set for service *ser1* from Figure 1 is as follows: IIR (*ser1*) = {(si1, c1), (si1, p1)}. Note that as previously described in Definition 4, a service interface cannot usually be connected to other types of explicit interfaces due to 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: $\mathbf{ISR}(s) = \{(e_1, e_2) \in R_s \mid R_s \subseteq (CC \cup CI \cup IC \cup II \cup PP \cup PH \cup HH \cup BPSBPS \cup CP \cup PC \cup CH \cup CBPS \cup BPSC \cup BPSI \cup PBPS \cup BPSP \cup BPSH \cup PI) \wedge e_1, e_2 \in (BPS_s \cup C_s \cup I_s \cup P_s \cup H_s)\}$.

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

Proceedings of the 2007 Australian Software Engineering Conference (ASWEC'07)
0-7695-2778-7/07 $20.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

DEFINITION 4.C (relationships between the service implementation elements of a given service and the service implementation 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):**

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

ii) outgoing relationships **(OR):**

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

From Figure 1: IR (*ser1*) = {(c3, ph)};
OR (*ser1*) = {(c2, c4)}.

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 the service interface si) by:

i) service incoming relationships **(SIR):**

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

ii) service outgoing relationships **(SOR):**

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

From Figure 1: SIR (*ser1*) = {(c4, si1)};
SOR (*ser1*) = {(p2, si2)}.

DEFINITION 5 (collaboration relationships between service-oriented design entities)

To capture the dynamic aspects of the system, a concept of a *collaboration* (***c***) has been defined. A collaboration captures elements that interact in order to achieve some desired functionality in response to *all possible* invocations of an operation ***o*** of some element ***e*** as follows: c = **<Param(o∈O(e)), CS>**, *where Param(o∈O(e))* represents parameters (inputs) to the operation *o* belonging to set of operations *O* of the generic element *e* as per Definitions 2 and 3; and *CS* is the set of all possible collaboration sequences *cs*. A collaboration sequence *cs* can be formally defined as: $cs = \langle SI_{cs}, BPS_{cs}, C_{cs}, I_{cs}, P_{cs}, H_{cs}\rangle$, *where* $\mathbf{SI_{cs} \subseteq SI, BPS_{cs} \subseteq BPS, C_{cs} \subseteq C, I_{cs} \subseteq I, P_{cs} \subseteq P, H_{cs} \subseteq H}$

This represents the set of interacting elements that achieve functionality exposed in ***o*** based on particular inputs. In terms of graph theory notation [13], collaboration sequence *cs* represents an open or closed walk starting at element ***e***.

3.3. Combined structure and relationships

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

DEFINITION 6 (SO System and Service)

− In the general case, a service-oriented system **SOS = <SI, BPS, C, I, P, H, R>**

− Given a system (*SOS*), a service **ser** = $\langle \mathbf{si_{ser}, BPS_{ser}, C_{ser}, I_{ser}, P_{ser}, H_{ser}, R_{ser}} \rangle$ *is a service of SOS if and only if* $si_{ser} \in SI \wedge BPS_{ser} \subseteq BPS \wedge C_{ser} \subseteq C \wedge I_{ser} \subseteq I \wedge P_{ser} \subseteq P \wedge H_{ser} \subseteq H \wedge R_{ser} \subseteq R \wedge R_{ser} \subseteq (IIR(ser) \cup ISR(ser) \cup IR(ser) \cup OR(ser) \cup SIR(ser) \cup SOR(ser)) \wedge (BPS_{ser} \cup C_{ser} \cup I_{ser} \cup P_{ser} \cup H_{ser} \langle\rangle ser)$

Given the above definitions, the *inclusion*, *union* and *intersection* operations for services can be defined as follows:

- ***Inclusion***: service $s = \langle si_s, BPS_s, C_s, I_s, P_s, H_s, R_s\rangle$ is said to be included in service $t = \langle si_t, BPS_t, C_t, I_t, P_t, H_t, R_t\rangle$ (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 $s = \langle si_s, BPS_s, C_s, I_s, P_s, H_s, R_s\rangle$ and $t = \langle si_t, BPS_t, C_t, I_t, P_t, H_t, R_t\rangle$ (notation $\mathbf{s \cup t}$) is the service $st = \langle si_{st}, BPS_s \cup BPS_t, C_s \cup C_t, I_s \cup I_t, P_s \cup P_t, H_s \cup H_t, R_s \cup R_t\rangle$, where interface si_{st} contains combined operations from si_s and si_t

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

Furthermore, *empty, disjoint, composite* and *atomic* services are 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 $\mathbf{SOR\ (s) \cup OR\ (s)} \neq \varnothing$ is said to be a composite service

- Atomic service: service **s** with $\mathbf{SOR\ (s) \cup OR\ (s)} = \varnothing$ is said to be an atomic service

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:

0-7695-2778-7/07 $20.00 © 2007 IEEE
IEEE COMPUTER SOCIETY

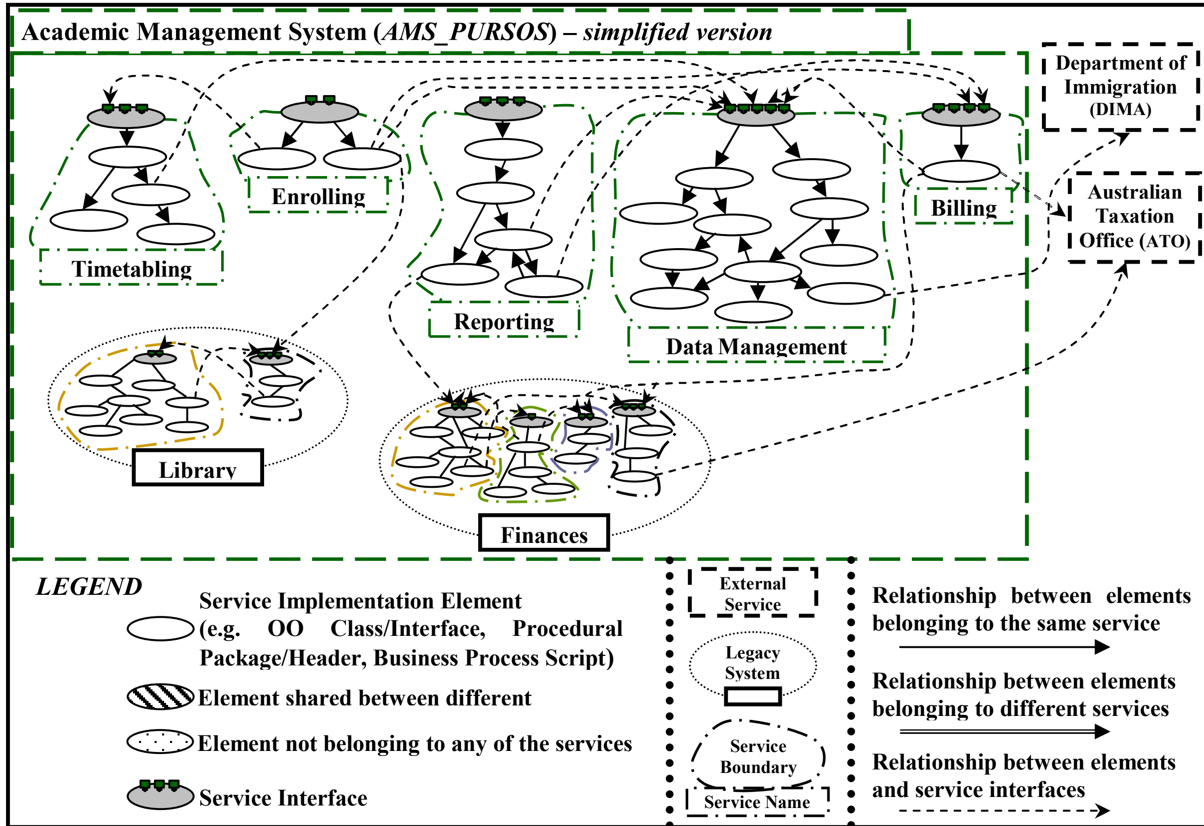

Figure 2. Example Pure Service-Oriented System Design

DEFINITION 6.1 (Partitioned SOS (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)*.

Formally, a system **PARSOS = <SOS, SER>** is a partitioned service-oriented system, only if **SOS = <SI, BPS, C, I, P, H, R>** is a service oriented system as per definition 6, and **SER** is a collection of services of **SOS** such that: $\forall \mathbf{bps} \in \mathbf{BPS}\ (\exists \mathbf{ser} \in \mathbf{SER}\ (\mathbf{bps} \in \mathbf{BPS_{ser}})) \wedge \forall \mathbf{c} \in \mathbf{C}\ (\exists \mathbf{ser} \in \mathbf{SER}\ (\mathbf{c} \in \mathbf{C_{ser}})) \wedge \forall\ \mathbf{i} \in \mathbf{I}\ (\exists \mathbf{ser} \in \mathbf{SER}\ (\mathbf{i} \in \mathbf{I_{ser}})) \wedge \forall \mathbf{p} \in \mathbf{P}\ (\exists \mathbf{ser} \in \mathbf{SER}\ (\mathbf{p} \in \mathbf{P_{ser}})) \wedge \forall \mathbf{h} \in \mathbf{H}\ (\exists \mathbf{ser} \in \mathbf{SER}\ (\mathbf{h} \in \mathbf{H_{ser}}))$

DEFINITION 6.2 (Pure SOS (PURSOS))

A system that is both partitioned as per definition 6.1, 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 6) is considered to be a *pure service-oriented system (PURSOS)*. The '*Academic Management System*' *(AMS_PURSOS)* shown in Figure 2 is an example of such a system. It consists of eleven fully independent services that communicate with other internal and external services strictly via service interfaces.

Formally, a system **PURSOS = <SOS, SER>** is a pure service-oriented system, if **SOS = <SI, BPS, C, I, P, H, R>** is a service oriented system as per definition 6 and **SER** is a collection of services of **SOS** such that: $\forall \mathbf{bps} \in \mathbf{BPS}\ (\exists \mathbf{ser} \in \mathbf{SER}\ (\mathbf{bps} \in \mathbf{BPS_{ser}})) \wedge \forall \mathbf{c} \in \mathbf{C}\ (\exists \mathbf{ser} \in \mathbf{SER}\ (\mathbf{c} \in \mathbf{C_{ser}})) \wedge \forall\ \mathbf{i} \in \mathbf{I}\ (\exists \mathbf{ser} \in \mathbf{SER}\ (\mathbf{i} \in \mathbf{I_{ser}})) \wedge \forall \mathbf{p} \in \mathbf{P}\ (\exists \mathbf{ser} \in \mathbf{SER}\ (\mathbf{p} \in \mathbf{P_{ser}})) \wedge \forall \mathbf{h} \in \mathbf{H}\ (\exists \mathbf{ser} \in \mathbf{SER}\ (\mathbf{h} \in \mathbf{H_{ser}})) \wedge \forall\ \mathbf{ser_i}, \mathbf{ser_j} \in \mathbf{SER}\ (\mathbf{ser_i} \cap \mathbf{ser_j} = \varnothing)$.

4. Customising the model – BPEL4WS

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 (http://xml.coverpages.org/xlang.html), IBM used Petri Nets with *Web Services Workflow Language* (www.ibm.com/software/solutions/webservices/pdf/WSFL.pdf), and BPMI.org developed the *Business Process Modelling Language (BPML)* from scratch (http://www.service-architecture.com/web-services/articles/business_process_modeling_language_bpml.html).

0-7695-2778-7/07 $20.00 © 2007 IEEE
IEEE COMPUTER SOCIETY

Subsequently, the *Business Process Execution Language for Web Services (BPEL4WS)* (http://www-128.ibm.com/developerworks/library/specification/ws-bpel/) was developed by a consortium of major software vendors including IBM and Microsoft. This is the latest in the series of BPM languages and arguably the most widely used, combining ideas from XLANG and WSFL specifications. The original BPEL4WS as well as its newer version '*WS-BPEL 2.0'* are currently in the standardisation stage at OASIS (http://www.oasis-open.org/)

Given the above, this section shows how the model presented in section 3 can be modified for the case of BPEL4WS, since a specific technology can 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 on the above constraints, the definition of the system and service structures (section 3 - Definition 1), are refined, as are the sets of *common, possible*, and *impossible* relationships (section 3 - Definition 4).

The rest of the definitions presented in section 3 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 **BPEL4WS Case** (System structure, Service structure)

- a *system* SYS = <SI, **BPEL**, C, I, P, H>, *where* SI = the set of all service interfaces in the system, **BPEL = the set of all BPEL scripts**, C = the set of all OO classes, I = the set of all OO interfaces, P = the set of all packages, H = the set of all package headers. This definition is similar to the generic one described previously.
- given a system (*SYS*), 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* $si_s \in SI \wedge \mathbf{bpel_s \in BPEL} \wedge C_s \subseteq C \wedge I_s \subseteq I \wedge P_s \subseteq P \wedge H_s \subseteq H$. Note: the ⊕ symbol indicates exclusive OR.

DEFINITION 4 **BPEL4WS Case** (relationships between different design artefacts in SOS systems)

The following relationships can exist in SOA systems employing *BPEL* scripts:

i) *Common* relationships R_c = <CSI ⋃ SIC ⋃ CC ⋃ CI ⋃ IC ⋃ II ⋃ PSI ⋃ SIP ⋃ PP ⋃ PH ⋃ HH ⋃ **BPELSI ⋃ SIBPEL**>, *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: $\mathbf{R = R_c \cup R_p}$.

5. Application

As stated previously, the design principles of SOC are yet to be well understood, and as such, there is a need for mechanisms allowing the measurement of structural properties of service-oriented designs in order to specify design guidelines and methodologies, as well as predict the quality of the final product.

Therefore, in order to provide an example of applying the model described in section 3, this section considers the derivation of two *coupling* metrics (where coupling is defined as a *measure of the extent to which interdependencies exist between software modules[9])* and their application to the example designs of Figure 2 and Figure 3.

The two example metrics presented below are part of an initial suite of SO design metrics (19 metrics in total) [23], which was derived using the model described in this paper. The metrics are intended to predict the quality characteristic of maintainability of service-oriented software in terms of *analysability*, *changeability*, and *stability* [15].

Example Metric 1: *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, $\boldsymbol{SPURF(SOS) = 1 - |IS(SOS)| / |SER|}$, *where* $\boldsymbol{IS}$ is the set of all the *intersected services* in the system $\boldsymbol{SOS}$, which can be expressed as: $\boldsymbol{IS(SOS)} = \{ser1 \mid ser1 \in SER \wedge \exists ser2 \in SER(ser1 \cap ser2 \neq \varnothing)\}$; and $\boldsymbol{SER}$ is a set of all the services of $\boldsymbol{SOS}$.

IEEE COMPUTER SOCIETY

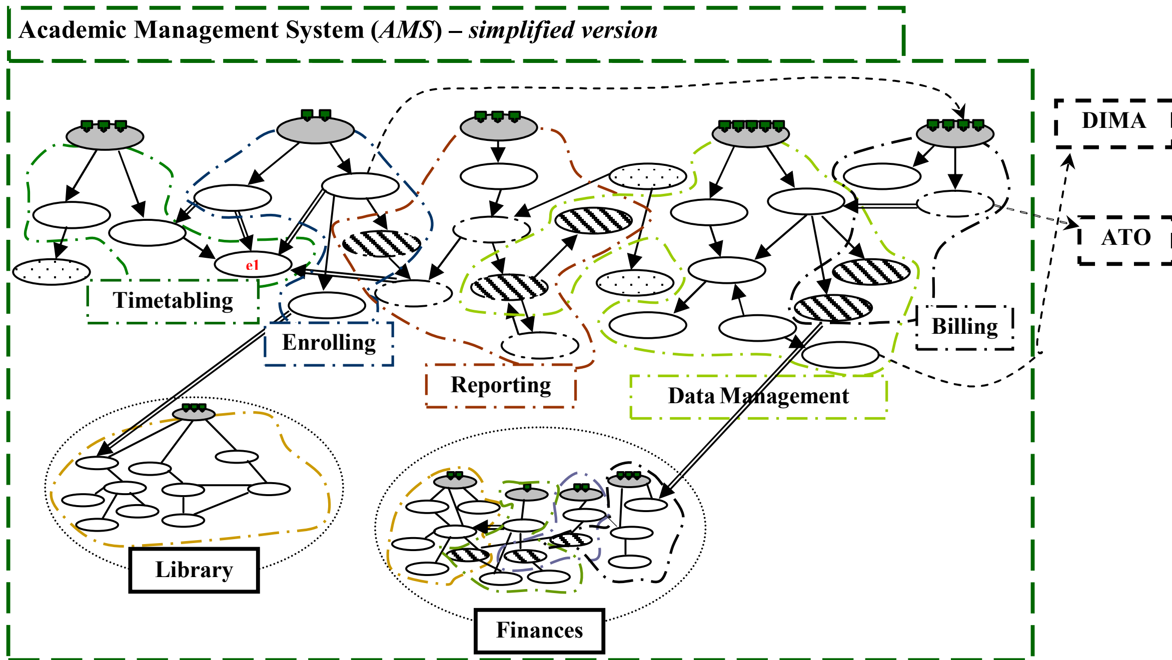

Figure 3. Example Non Pure Service-Oriented System Design

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 Figure 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 [4] 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 Figure 3 as reflected by *SPURF* (***AMS***) = 0.1.

Example Metric 2: *Weighted Extra-Service Incoming Coupling of Element (WESICE)*

WESICE for a given service implementation element ***e*** of a particular 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) =*** $|\{(e, e1) * WeightFactor | \{e, e1\} \in IR(s)\}|$, *where* ***IR(s)*** is the set of inter-service incoming direct relationships for service ***s*** (section 3, 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) relationship type since the coupling is expected to be 'stronger' in the former case [23].

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 work or refactoring. For example, WESICE (***e1***) = 9 in the design shown in Figure 3, meaning that this is a potential design problem that should either be addressed prior to implementation or refactored during maintenance.

6. Conclusions

This paper has presented a formal model covering design artefacts and associated relationships of service-oriented systems. This model formalises structural concepts of service-oriented development, which promotes a better understanding of SOC, thus potentially supporting reasoning and decision making during the design and implementation of such systems. Furthermore, the model has laid the foundation for the derivation of software metrics for measuring structural properties of SO designs by allowing such metrics to be defined and validated in a precise mathematical manner. Therefore, to demonstrate this application of

0-7695-2778-7/07 $20.00 © 2007 IEEE
IEEE COMPUTER SOCIETY

the model, two example metrics from an initial suite of service-oriented coupling metrics were presented.

Additionally, 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 could be added in the future with little change to the basic model.

In future work, the authors plan to further evaluate the model by using it to produce abstract representations of real-life large-scale SO system designs. Such representations will than be measured by the proposed metrics in order to identify important design characteristics influencing the quality of SO software products. This should lead to the identification of design guidelines, best practices and ultimately a complete methodology for the development of service oriented systems.

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

7. References

[1] M. Acharya, A. Kulkarni, R. Kuppili, et al., "SOA in the Real World – Experiences," presented at 4th Intl. Conference on Service Oriented Computing, 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, et al., *Web Services: Concepts, Architectures and Applications*. Heidelberg, Germany: Springer-Verlag, 2004.

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

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

[6] D. Berardi, F. De Rosa, L. De Santis, et al., "Finite State Automata As Conceptual Model For E-Services," *Journal of Integrated Design and Process Science*, vol. 8 (2), 2004.

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

[8] L. C. Briand, J. Wust, J. Daly, et al., "Exploring the relationship between design measures and software quality in object-oriented systems," *Journal of Systems and Software*, vol. 51 (3), 2000.

[9] 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.

[10] M. Endrei, M. Luo, P. Comte, et al., *Patterns: Service-Oriented Architecture and Web Services*: IBM Redbooks, 2004.http://www.redbooks.ibm.com/redbooks/SG246303/

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

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

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

[14] E. Gamma, R. Helm, R. Johnson, et al., *Design patterns: elements of reusable object-oriented software*. Reading, Mass.: Addison-Wesley, 1995.

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

[16] 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.

[17] 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.

[18] S. Morasca, "Measuring Attributes of Concurrent Software Specifications in Petri Nets," presented at 6th Intl. Symposium on Software Metrics, Florida, USA, 1999.

[19] M. P. Papazoglou and A. D. Georgakopoulos, "Service-Oriented Computing," *Communications of the ACM*, vol. 46 (10), pp. 24-28, 2003.

[20] J. Pasley, "How BPEL and SOA are changing Web services development," *Internet Computing, IEEE*, vol. 9 (3), pp. 60-67, 2005.

[21] 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 Intl. Workshop on Modeling Inter-Organizational Systems, Ayia Napa, Cyprus, 2005.

[22] 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.

[23] M. Perepletchikov, C. Ryan, K. Frampton, et al., "Coupling Metrics for Predicting Maintainability in Service-Oriented Designs," presented at 18th Australian Conference on Software Engineering (ASWEC 2007), Melbourne, Australia, 2007.

[24] P. Rossi and G. Fernandez, "Definition and Validation of Design Metrics for Distributed Applications," presented at Ninth International Software Metrics Symposium, Sydney, Australia, 2003.

[25] H. Schmidt, "Trustworthy components: compositionality and prediction," *Journal of Systems and Software*, vol. 65 (3), pp. 215-225, 2003.

[26] M. P. Singh and M. N. Huhns, *Service-Oriented Computing: Semantics, Processes, Agents*. West Sussex, England: John Wiley & Sons, 2005.

[27] 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

IEEE COMPUTER SOCIETY