

A Promise Theory View on the Policies of Object Orientation and the Service Oriented Architecture

Demissie Aredo, Mark Burgess and Simen Hagen

Oslo University College, P.O.Box 4, St Olavs Plass, 0130 Oslo, Norway

Abstract

Object orientation (OO) and Service Oriented Architecture (SOA) are two approaches to software design. By modelling these using Promise Theory we show that we can easily accomodate SOA principles into program modelling. We also show that promise theory can help to resolve ambiguities in OO design, and we resolve a case in which UML semantics do not offer sufficient guidance to determine a natural structure in a program. Our approach offers some simple clarity on the various meanings of inheritance policy and the Liskov substitution principle.

Key words: Information modelling, object orientation, service oriented architecture

1 Introduction

Object Orientation (OO) is a policy for the organization of software code[1,2], based on a behaviourist view of software design and a roughly standardized set of tools and mechanisms. It is realized through a number of programming languages and the Unified Modelling Language (UML), and has attained considerable popularity for its ability to separate issues and build hierarchies of customizable and reusable data-types. Its strict application also has certain weaknesses, as is testified to by their inability to avoid code duplication in certain circumstances, such as product line variability. This has led interest in complementary policies like *aspect orientation*[3], as well as the continued popularity of procedural and functional programming languages.

The Service Oriented Architecture (SOA) is a request-based architecture in which services are performed on a voluntary basis, often by different organizations or companies who provide the servers[4]. In spite of partially overlapping

philosophies, the two architectures are somewhat different. Object orientation assumes a flow of logic in a program in a hierarchy without option to decline. In the SOA, no single agent has the capability (or even the right) to demand service from another. Exchanges are purely based on voluntary cooperation. The architecture is built on a potentially unreliable communication channels, often owned by a third party, and permissions and service levels are granted with the grace of the server administrator, coded in a Service Level Agreement (SLA).

In this paper, we follow Alexandrescu in considering OO and SOA interfaces to be different policies for program writing[5]. We do this by embedding programming design decisions within a framework of policy declarations. This will allow us to see the structure and details of the interfaces and their implications for correctness and security. All policies can then be understood as aggregations of promises. Program modelling is normally carried out today using the Unified Modelling Language (UML), but UML starts from the assumption that OO is an appropriate model for a program. We pose the question differently: can we describe the essential interfaces in behaviour driven models using a *declarative policy framework*? And further, does this reveal anything about which structural policies are well suited to solving a given problem? We use the idea of *promises*[6–8] as a model for distributed *policy*.

The plan for our paper is as follows. We begin by defining the basic notions from promise theory that are relevant to model the architecture of a computer program. In particular we describe how autonomous agents promise to agree on basic commonalities such as data-types etc., and we go on to describe how agents can be classified in terms of the promises they make. Next we introduce the main features of the Service Oriented Architecture, whose philosophy is already close to that of promise theory, noting the role of ontologies where data-types are not consistent. Finally we discuss how one can reproduce familiar ideas about Object Orientation, such as inheritance, with the right promises and hence in terms of a Service Oriented model.

2 Program objects and their organization

We limit our ambitions in the present paper by considering only structures and interfaces that describe program organization. This includes OO concepts like *data types* and *classes*.

From a structural viewpoint, a program is a collection of *data types* and *methods* which interact and manipulate instances of the types. Data-types (DT) are syntactic labels which achieve standardized processing. In a typical OO design, data types are *common knowledge* to all parts of a program, starting

from the compiler and its libraries. In the SOA, however, different services are coded and implemented on different servers, in possibly different domains, with potentially incompatible languages and even different data-type schemas. These can be controlled by different parties with possibly different interests. Clients must always deal with the problem of matching interfaces[9,10]. Recently Bergstra and Ponse have given a formulation of interfaces based on groups for the symmetry of give and take[11]. Our promise model agrees nicely with their viewpoint.

3 Promise theory

Promise theory is a language for discussing policy in networked environments. We define policy to be an accumulation of predecided characteristics or *behaviours*, some of which are context dependent, i.e. a *specification* of decisions made within the framework of a model. A promise network is built up from entirely autonomous elements that have no authority or control over one another, and no *a priori* common knowledge[6].

Definition 1 (Agent node) *An agent is an autonomous entity in a graph. Autonomy implies that each node makes all of its own decisions. No node in the graph can be coerced or may make a demand of any other. No agent can require any property of another.*

Both promisers and promisees are referred to as *agents* but these should not be confused with the notion of agents in the Actor Model[12], or the concept autonomous agents as used in Multi-Agent systems[13,14] where ‘autonomous’ simply means that agents work independently but are commanded from an outside source. We make no judgement on whether the agents in a program work serially or parallel here.

Definition 2 (Promise) *A promise is an autonomous declaration of behaviour by an agent. It is represented as a labelled edge in a graph from a promiser to a promisee. Each promise thus has a sender S and a receiver R . The subject of the promise b is called the promise body. We use the notation:*

$$S \xrightarrow{\pi:b} R. \tag{1}$$

An agent’s policy, according to this view, is now simply a collection of its promises. The entire promise graph forms a *distributed policy*. In this paper, we assume that both the sender and the recipient alone are aware of promises.

The promise body b is typically a doublet $\langle \tau, \chi \rangle$, where τ is a *promise type*, and χ is a constraint which indicates the subset of possible values within the domain of τ that the agent promises to keep to. e.g. τ is a promise to contain an integer value, and χ says that it must lie in a given subset between 12 and 14. We therefore have a simple set-theoretic notion of a promise which lives in a network of independent agents. See refs. [7,8,15] for more about promises.

3.1 Categories and types of promise

Promises require their own notion of *type*, which we strive not to confuse with the *data-types* of programming languages. Promise types and program data-types are *a priori* unrelated, but we shall relate them later. Additionally, there are two distinct categories within each type of promise body: these are service (give) and acceptance (take) promises. It is sometimes convenient to define a third type called the coordination promise[6] which can be derived from the others.

- A *service promise* body, whose body is denoted by b or $+b$, is the basic type of promise. It denotes a restriction of behaviour by the promising agent (as if this restriction were a service to the promisee):

$$S \xrightarrow{\pi:b} R \text{ or } S \xrightarrow{\pi:+b} R \quad (2)$$

- A *usage or acceptance promise*, denoted by $U(b)$ or $-b$, is the promise to receive and use a service b promised by another agent.

$$S' \xrightarrow{\pi:U(b)} R' \text{ or } S' \xrightarrow{\pi:-b} R' \quad (3)$$

- A *coordination or subordination* promise, denoted by $C(b)$, is the promise to do the same as another agent with respect to a promise body b . This is not a primitive type, but we denote it by the shorthand:

$$S'' \xrightarrow{\pi:C(b)} R'' \quad (4)$$

involves that R'' informs S'' about its actions with respect to promises of type b , and the receipt and usage of that information by S'' .

When two promises originating from an agent are incompatible, they cannot be realized physically at the same time. We write

$$A_1 \xrightarrow{\pi:b_1} A_2 \# A_1 \xrightarrow{\pi:b_2} A_3 \quad (5)$$

If $A_2 = A_3$, we may omit the agents and write $b_1 \# b_2$. For example, a single agent promising two promises of the same type but different constraints b_1 :

$(\tau, v = 2)$ and $b_2 : (\tau, v = 3)$ is not realizable. Breaking a promise is not the same as not keeping a promise; it is an explicit contradiction.

Definition 3 (Broken promise) *A promise to an agent is broken if one promise is contradicted by another promise. We define a broken promise as the promising of exclusive promises*

$$A_1 \xrightarrow{\pi:b_1} A_2, A_1 \xrightarrow{\pi:b_2} A_2 \text{ with } b_1 \# b_2 \quad (6)$$

Clearly promising b and $\neg b$ would be incompatible, and hence would constitute a broken promise.

4 Knowledge and Scope

Scope is a key feature of programming models and an important difference between SOA and OO, hence we describe it from a theory of promises which begins with absolute limitation of scope (private knowledge). Thus we develop the basic ideas that allow us to discuss SOA collaboration and OO properties in SOA.

Definition 4 (Agent Knowledge) *Let $K_{A_i}^v$ be a set of data values that are internal to agent node A_i . We call $K_{A_i}^v$ the knowledge of agent A_i about topic or value v . This knowledge is not available to any other agent, unless its value is promised to them.*

We use a notation K^v, K_1^v, K_2^v as follows. Let $\mathcal{K}$ be the set of all possible variables or subjects that can be known, and consider knowledge about a specific set of values or subject $K^v \in \mathcal{K}$. We use K_1^v to mean agent A_1 's knowledge of this subject v , and K_2^v to be A_2 's knowledge about K^v etc. Thus different agents (i) can have different knowledge about the same subject (v). We shall use this, in particular, to discuss agents' varying type schemas.

Definition 5 (Consistent Knowledge) *Two agents S and R , with knowledge K_S^v and K_R^v , can claim to have consistent knowledge about k^v iff they each can verify the equality of a subset of their knowledge $k_S^v = k_R^v$ where $k_S^v \subseteq K_S^v$ and $k_R^v \subseteq K_R^v$.*

To simplify notation henceforth, we pick a single subject and suppress the v superscripts.

Theorem 1 (Consistent Knowledge) *Let S and R be any two agent nodes, and let $\pi : k$ be a promise body that implies communicating knowledge k . Nodes*

S and R have consistent knowledge k iff

$$S \xrightarrow{\pi: +k_S} R \quad (7)$$

$$R \xrightarrow{\pi: -k_S} S \quad (8)$$

$$R \xrightarrow{\pi: +k_R} S \quad (9)$$

$$S \xrightarrow{\pi: -k_R} R \quad (10)$$

$$S \xrightarrow{\pi: (k_S = f(k_R, k_S))} R \quad (11)$$

$$R \xrightarrow{\pi: (k_R = f(k_R, k_S))} S \quad (12)$$

where $f(a, b)$ is some function of the original values that must be agreed upon.

Proof 1 By definition 4 the knowledge k_S of any agent S is only available to another agent node R iff its value is promised as in eqns. (7) and (9). The knowledge is not certainly received until the agents promise to accept the promised values, as in eqns. (8) and (10). Now both agents know each others' values. To make their knowledge consistent, by definition 5 they must now promise each other to make their values equal. This can be done through an intermediary function $f(a, b)$ which signifies that an equilibrium is formed by some unspecified consensus.

Consistency of knowledge is a strong concept which requires the notion of verification; at the same time it is clear that knowledge might be both available and consistent in a service oriented network, even if one cannot actually verify that consistency, e.g. if it is disseminated from a single source.

Let K_S be the knowledge of agent S . Knowledge K_S is said to be relayed from a source S to a receiver agent R iff

$$\begin{aligned} S &\xrightarrow{\pi: K_S} R \\ R &\xrightarrow{\pi: U(K_S)} S \end{aligned} \quad (13)$$

The fact that knowledge has been passed on and received does not mean that it has been *assimilated* by the agent, i.e. that it will replace its own knowledge about K with this new version. For example, in peer to peer systems, such as media distribution, relay stations, routers, data are frequently relayed in this way without necessarily being retained. A new version of a song or movie could be passed on to others without replacing a local authoritative version used by the local system.

Definition 6 (Assimilated knowledge) Knowledge K_S is said to be *assimilated* by a receiver agent R from a source S iff

$$\begin{array}{ccc}
S & \xrightarrow{\pi:K_S} & R \\
R & \xrightarrow{\pi:U(K_S)} & A_1 \\
R & \xrightarrow{\pi:(K_R=K_S)} & A_2
\end{array} \tag{14}$$

where A_1, A_2 could be some unspecified agents. We shall assume here that $A_1 = A_2 = S$ for simplicity.

An agent does not know K unless it is either the source of the knowledge, or it has promised to accept and assimilate the knowledge from the source. It does not matter to whom the promises to use and assimilate the data are made, as long as they are kept, though it seems natural to make the promises to S so that S can verify the transmission.

Knowledge K can be passed on from agent to agent with integrity, but we must distinguish between agents that end up with a different picture of the information as a result of their local policies for receiving and relaying their knowledge.

- (1) Accepted from a source, ignored and passed on to a third party intact.

$$\begin{array}{ccc}
A_1 & \xrightarrow{\pi:K_1} & A_2 \\
A_2 & \xrightarrow{\pi:U(K_1)} & A_1 \\
A_2 & \xrightarrow{\pi:K_1} & A_3
\end{array} \tag{15}$$

Note that the agent A_2 does not assimilate the knowledge here by making its own version K_2 equals to K_1 , it merely passes on the value as hearsay.

- (2) Accepted from a source, ignored and local knowledge is then passed on to a third party instead.

$$\begin{array}{ccc}
A_1 & \xrightarrow{\pi:K_1} & A_2 \\
A_2 & \xrightarrow{\pi:U(K_1)} & A_1 \\
A_2 & \xrightarrow{\pi:K_2} & A_3
\end{array} \tag{16}$$

Here the agent A_2 accepts the information but instead of passing it on, passes on its own version. The source does not know that A_2 has not relayed its data with integrity.

- (3) Accepted and assimilated by an agent before being passed on to a third party with assurances of integrity.

$$\begin{array}{ccc}
A_1 & \xrightarrow{\pi:K_1} & A_2 \\
A_2 & \xrightarrow{\pi:U(K_1)} & A_1
\end{array}$$

$$\begin{aligned}
A_2 &\xrightarrow{\pi:K_2=K_1} A_1 \\
A_2 &\xrightarrow{\pi:K_2=K_1} A_3 \\
A_2 &\xrightarrow{\pi:K_2/K_1} A_3
\end{aligned} \tag{17}$$

A_2 uses the data from A_1 , assimilates it ($K_2 = K_1$) and promises to pass it on (conditionally K_2/K_1) if it receives K_1 . It also promises to both involved parties to assimilate the knowledge. Only in this case does the knowledge K become common knowledge if one continues this chain.

The first two situations are indistinguishable by the receiving agents. In the final case the promises to make $K_1 = K_2$ provide the information that guarantees consistency of knowledge throughout the scope of this pattern. We can now define scope.

Definition 7 (Scope of common knowledge) *Let S be a set of source agents with consistent knowledge K , and let the set $\mathcal{A}(X, K)$ mean the set of nodes that have assimilated knowledge from a set of agents X . The scope $\mathcal{S}(K)$ of K is the union of S with all agents that have assimilated knowledge originating from S , i.e.*

$$\mathcal{S}(K) = S \cup \mathcal{A}(S \cup \mathcal{S}(K), K) \tag{18}$$

The simplest way to achieve common knowledge is to have all agents assimilate the knowledge directly from a single source agent. This minimizes the potential uncertainties, and the source itself can be the judge of whether the appropriate promises have been given by all agents mediating in the interaction. The scope is then simply a cooperative promise group in sense of ref. [7]. This is the common understanding of how a network *directory service* works. Although simplest, the centralized source model is not better than one in which data are passed on epidemically from peer to peer. The problem then is simply in knowing the boundaries of scope.

From these definitions, we can see that agents can have consistent knowledge from an authoritative source, either with or without centralization, and scope can be controlled using promise theory to describe both OO and SOA.

5 Promises for modelling software

If we adopt the promise theory viewpoint, i.e. that each data object and method is an autonomous agent, then a program specification incorporates a

large number of promises between its components, and these promises reflect the policy used in designing and implementing the program.

5.1 Definitions

To model programs as freely cooperating constellations of agents, we can now employ the following association:

Agent node $\leftrightarrow$ Data value, Object, Processing element
 Promise $\leftrightarrow$ Assignment, Interface property, Service

The success of any program can thus be thought of as relying on a number of promises between agents, over which we have no outside control. We call this *voluntary cooperation*[16,8].

Let us restate some well known programming concepts in terms of promises.

Definition 8 (Program Method) *A processing element in a program that can optionally receive input and optionally generate output on demand. A method's behaviour is specified by its internal processing code (within the body).*

We do not need to make this definition more specific here. We assume that readers have a common understanding of what a method is.

Definition 9 (Service) *A service specification is a promise, from an agent S to some unspecified agent A (denoted $S \xrightarrow{\pi:b} A$). The promise body constrains A to behave within certain limits (called its Service Level Promise).*

Note that services and their specifications are indistinguishable before execution of a program begins; so, from a architectural viewpoint, the prediction of future behaviour implied in a promise is indistinguishable from the service itself.

Definition 10 (Data Object) *A data object is a program element that can optionally receive input to set its value and optionally generate output to inform of its value.*

It is easy to see that data objects are trivial methods (also called identity functions). Let us now make the identifications with promises. In promise theory, objects map to agents, but agents do not have types as data values do in programs. It is the promises they make which have types.

Lemma 1 (Methods are services) *A program method is a collection of services.*

Consider the services $S_1 \xrightarrow{\pi:\text{value}} A$ for some A (to provide a data value to some recipient), and $S_2 \xrightarrow{\pi:U(\text{value})} A$ (to receive a data value from some source). If there is at least one agent S_2 then information can be received on demand. If there is at least one agent S_1 , a value will be returned. The composition of zero or more of these agents hence forms a constellation that can optionally exchange input/output based upon its internal decisions.

Lemma 2 (Data objects are services) *A program object which supplies a data value is a pair of optional services: one to accept changes in value and one to inform of its value.*

The proof of this is trivial: one can always view a data object as a method which returns an internal value. This is even common practice in Object Oriented programming. This leads to the straightforward equivalence of data objects and methods, within the promise framework. This equivalence will allow us to build the concept of classes from bundles of promises in section 8. The programming data-type of a value has not yet been addressed here; but it is straightforward to note that it can be included in the type of the promise (see below).

5.2 Examples

We pause to illustrate a few well-known programming idioms using the idea of autonomous agents and promises, i.e. a service oriented view.

Example 1 *A variable assignment $\mathbf{x=y}$ involves a promise to yield the value v held at b to a : as well as a promise by a to use or accept that value from b :*

$$\begin{array}{l} y \xrightarrow{\pi:b} x \\ x \xrightarrow{\pi:U(b)} y. \end{array} \quad (19)$$

The data-type of the value is implicit in the type of the promise body b and $U(b)$. One can simply identify the promise type of b with the data value's identifier together with its data type: $\tau(b) \leftrightarrow \langle \text{id}, DT \rangle$. The use-promise has b as its value, meaning that it promises to accept any value in the domain $\chi(b)$, e.g. this might map to values of data-type int , etc. In a general service architecture, the client x could refuse to accept certain values as being illegitimate. Similarly in a strongly typed language, such as Ada or Pascal, a variable could refuse to accept a value that falls outside a specific range.

Example 2 *In figure 1(a) we capture the essence of the expression $a = f(b)$, in which node b contains a value v which is processed with a function f . In*

(b) we use indirection (pointers) to find out either the location of a value $p = \&b$; $a = *p$ or the location of the variable $p = \&b$; $a = p$. These two alternatives are represented as different promise types. In (c) a linked list is represented as a chain of linked pointer promises next to a set of value nodes. Partner nodes promise coordination so they belong to a compound object. An additional promise v to an external coordinator adjudicates this membership.

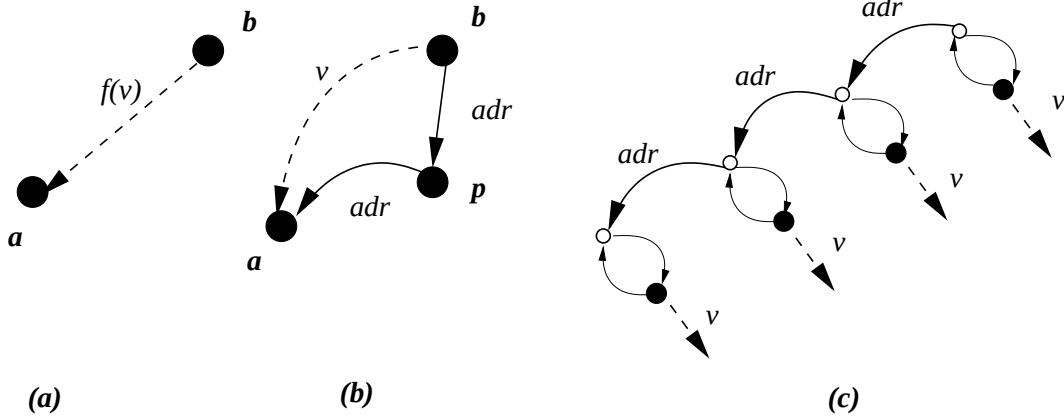


Fig. 1. (a) $a = f(b)$, (b) $a = *p$, (c) A linked list.

6 Classes: roles and promise bundles

We now turn to patterns within the promise graph, and define the concepts of a bundle and a role of promise types. We shall use this concept below to define a *class* in the object oriented sense. We shall find that roles are representative of SOA and bundles are representative of OO.

Definition 11 (Promise bundle) *Let $S, R \subseteq A$ be arbitrary subsets of equal dimension d , from the set of agents. A promise bundle is any collection of promises made (one-to-one) from agents in S onto agents in R . We denote this*

$$S \xrightarrow{\Pi:B} R \equiv s_1 \xrightarrow{\pi:b_1} r_1, \dots, s_d, \xrightarrow{\pi:b_d} r_d. \quad (20)$$

where Π denotes a bundle of promises and B is the collection of promise bodies $b_1, b_2, \dots, b_d$.

An important concept in the promise graph is that of a *role*. To some extent we shall be able to identify promise-roles with the natural set of classes in an object oriented design. However, roles are more general than OO-classes. In promise theory, roles are associated both with *types* of promises τ and topology. In this respect, we can associate a collection of agents with a role,

given that they make (or use) the same set of promise types. In other words, we are guided by the typed structure of the graph alone.

Definition 12 (Role) *Let $\mathcal{A}$ be a set of autonomous agents. The family of sender/receiver subsets $S, R \subseteq \mathcal{A}$ in a promise graph, which take part in promises with body types $t(B)$, define B -roles. Any subset S or R in the graph is said to play a role of (sender/receiver) with body types $t(B)$ in the promise graph. Bundles with equal types and different constraints have equal roles.*

Note that promise roles are identified empirically, not pre-decided by design.

Example 3 *The set of all agent nodes $W \subseteq \mathcal{A}$ that promises to provide web service to any agent $R \in \mathcal{A}$:*

$$W \xrightarrow{\pi:\text{web}} (R \in \mathcal{A}) \quad (21)$$

plays a role, which we can call ‘web server’. This is called a role by association, since the members of W are not connected in any other way than they make the same promise. The set of nodes $C \subseteq \mathcal{A}$ that is promised web service by any agent:

$$(S \in \mathcal{A}) \xrightarrow{\pi:\text{web}} C \quad (22)$$

plays a role, which we can call ‘web client’. This is called a role by appointment, since the members of C are pointed to by a promise of the same type.

We refer readers to [7,8,15] for more information about promises and roles.

7 SOA concepts

The Service Oriented Architecture (SOA) is an informal set of principles for organizing a distributed web-based application. The basic concepts in the SOA are already quite close to the spirit in which Promises Theory builds its view of systems. The agents involved are autonomous (individually managed), they are loosely coupled, i.e. they cooperate voluntarily. Several servers could implement the same service, or services could be distinguished by their particular service provider. Thus the concept of promise-roles offers a natural classification of services in the SOA. Three qualities are mentioned in connection with SOA:

- Exposing software interfaces through web services.

- System components communicate over some form of standard structure (usually XML based).
- Asynchronous messaging transfers documents between different systems.

These concepts are essentially trivial except for the first. The first is non-trivial because web servers are autonomous agents, in the sense of promise theory, i.e. they have no *a priori* consistent knowledge. In addition to these points, we add that, in the SOA, there must be a “Service Level Agreement” (an agreed set of promises), which describes the availability and service quality of the services provided. Service is taken for granted in OO.

An important aspect of SOA systems, however, is that of an *ontological map*[9]. This is an interface that maps inequivalent typing standards between agents in a peer to peer fashion. The SOA potentially lacks a common data model. If parts of a program are distributed and designed by quite different developers, they will not necessarily share a common set of data types or interpretations. The Semantic Web project attempts to define data semantics for web services using a variety of languages[17]. However, promises should not be confused with Resource Description Framework (RDF)[17] or Reification algebra[18] as featured in discussions of the Semantic Web. When different agents communicate they require either a common knowledge of data-types or an *ontological map* to relate them.

Definition 13 (Ontological map) *A mapping from one data-type schema s_1 to another s_2 such that a maximum number of promises made by s_1 are kept by s_2 .*

Proposition 1 (Consistent type requires a type service) *Let O be a collection of type definitions, and let DT be an agent who knows these (a type arbitrator). Then, a group of agents G knows O consistently iff it is consistent knowledge. Hence there must be a source for this knowledge, which we call a type server.*

The proof follows from theorem 1. The type service, like a name or directory service, might be centralized or decentralized, as in the theorem.

As we know from the consistent knowledge theorem, there are only two ways to satisfy this closure requirement. Either one appoints a subset of servers who have dedicated knowledge and who agree to coordinate that knowledge and serve it to the rest (authoritative centralization); or, every agent interacts bilaterally with every other agent inside the set A (uniform peers). If two separate regions establish their own type servers, as in fig. 2, the interactions between them must be coordinated as indicated by coordination promises $C(O)$, i.e. they must agree to make the same promises about data-types.

We can now state the requirements for distributed interactions in SOA.

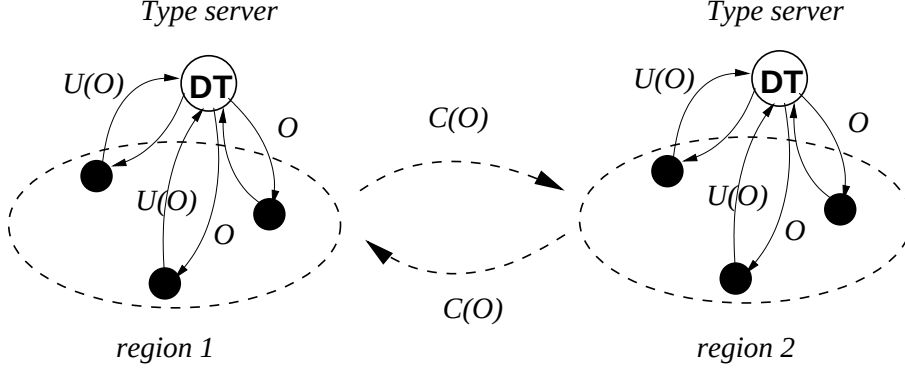


Fig. 2. Different data-type regions, served by independent type-servers. Data-type regions can cooperate (share) with one another. Each corresponding element of a class could use a different data-type server, as long as sender and recipient's definitions are coordinated.

Lemma 3 (Interface requirement) *Consider a client agent C and a server agent S and let q be a data value sent as a request to a server and v be the value of the reply. A service oriented interaction requires promises:*

$$DT \xrightarrow{\pi:\text{schema}} C \quad (23)$$

$$C \xrightarrow{\pi:U(\text{schema})} DT \quad (24)$$

$$S \xrightarrow{\pi:U(q)/U(\text{schema})} C \quad (25)$$

$$S \xrightarrow{\pi:v=f(q)/(q \& \text{schema})} C \quad (26)$$

to exchange a service $v = f(q)$.

The data-type server (DT) must promise to deliver information about its data model schema (types etc) to the client. Clearly it is possible that $DT = S$. The client promises to use that schema. The server promises to use any request q from the client if the client promises to obey the schema. Finally, the server promises to return the value of $f(q)$ if it receives q .

- (23) and (24) imply we have an ontological map.
- (25) ensures that C obeys the data-model, S uses parameter q .
- (26) ensures that S returns $v = f(q)$ provided q is made available in an acceptable data type.
- Hence $S \xrightarrow{\pi:v=f(q)} C$ is a method specification.

8 OO concepts

We are now in a position to consider how to model OO concepts in promise theory and hence in the SOA. We consider three properties:

- Aggregation of data types and methods into ‘classes’. This usually implies some philosophy for abstraction.
- Inheritance of classes.
- Sub-type polymorphism. The ability for sub-classes to override parent classes when referred to by name.

8.1 Data types

A compound type like a ‘record’ or ‘struct’, in a modern programming language, is simply a bundle of primitive data-types. In object terms, an instance of one of these compound types is a bundle of values. A class can additionally include methods as members. Using the promise theory view of the equivalence of values and methods with services (lemma 1), we can now reinterpret classes as collections of promise bodies from similar promise bundles.

It is straightforward to map the structured class name tuples in fig. 3 to a flat promise type space. For instance, from the figure:

$$\begin{aligned}\langle XYZ, type1, name1 \rangle &\rightarrow \tau_1, \\ \langle XYZ, type2, name2 \rangle &\rightarrow \tau_2, \\ \langle XYZ, type3, method(input) \rangle &\rightarrow \tau_3/\tau_4 \\ &etc.\end{aligned}\tag{27}$$

The hierarchical structure is not relevant to us; we only need to uniquely identify the members.

At this point, it is trivial to model *class instances* or objects as bundles of promises that are isomorphic (see fig. 3).

8.2 Classes and ‘natural’ roles

One of the controversial aspects of Object Oriented modelling is the idea that classes model real world situations and therefore lead to simpler programs. Let us take a different view, motivated by the idea of promises. Suppose we describe a program in terms of a set of promises it must keep, observing the

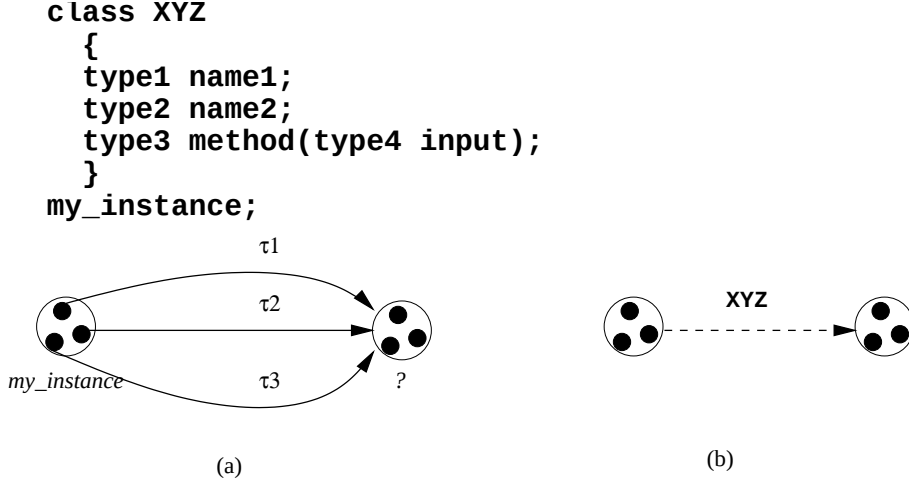


Fig. 3. A class instance is a promise bundle with a unique pattern.

principles of parsimony and non-overlap of promise types as far as possible: let us ask, is there a natural set of promise patterns that emerge in the promise graph of the program? Such basic patterns might be a natural set of classes for the program, or at least form a minimal *spanning set*. Let us then assume that we are operating in a region of common knowledge about data types.

Definition 14 (Class-like promise body) *Any unique collection of promise bodies $c \equiv \{b_1, b_2, \dots\}$, belonging to a promise bundle.*

A class-like bundle is a collection of promise bodies, i.e. it is missing the sender and the receiver. It is thus an abstract compound type.

Lemma 4 (Class-like bundles) *A class-like bundle is both a set of agents S involved as sender, and a set of receiver agents R together with a class-like promise body $(S \xrightarrow{\Pi; \mathcal{G}} R)$, iff each pair $(r, s) \in (R, S)$ agrees to use a common data-type definition for each $b \in c$ in the promise bundle.*

Notice that, even if the data types do not match, two apparently different class-like bundles in different parts of a program might in fact be equivalent set of promises between their respective agent sets, even if the class-like bundles do not match in terms of their data-type schemas. This would mean that there were inequivalent ways of compiling a data-type schema into the same promised behaviour, and would presumably be a sign that there existed a suitable ontological mapping between the class-like bundles.

In other words, class-like instances form distinguishable patterns of promises, that occur possibly several times in the graph and they must agree on the basic type-alphabet for their pattern matching, on a per-promise basis. A class-like bundle could now naturally be associated with an OO class, and the OO-class is thence the role belonging to a promise bundle. However, we must remember that the naming of classes in OO is a user policy decision, not a requirement.

Thus, we claim that while the roles lead us to a ‘natural’ set of classes, there is no obligation for any programmer, agent or analyst to adopt these as the actual set of OO classes. What we have identified is a natural ‘spanning set’ of class containers. This emphasizes that OO is a policy, not a necessity.

8.3 *Inheritance and usage*

The concept of inheritance is of central importance in Object Orientation, for philosophical reasons if nothing else. It is both used and explained using a number of different interpretations[19–21]. We can use promises to restate these meanings more clearly, though we cannot claim that our interpretations are those intended by other authors. We shall consider three such meanings:

- *Class 2 extends (adds to) class 1:*
When a child class extends another parent class, it adds values and methods that were not present in the parent class.
- *Class 2 substitutes (replaces) class 1*
When a child class substitutes a parent class, it replaces it completely with a new implementation. In the case of an abstract parent class it might, in fact, provide the first proper implementation.
- *Class 2 specializes (overrides) class 1:*
A subset of values and methods in the parent class can be replaced with new implementations. In the limit of complete overriding, this becomes the same as substitution.

Programming languages generally lump all three inheritance meanings together in a general construct, so an inheritance policy might imply several of these at the same time (e.g. one often extends a parent class into a number of child classes which are then thought of as specializations – here we would call them mutually exclusive extensions). We wish to be somewhat clear about these and therefore keep them separate. In addition to this there is *usage* or *delegation* in which a class “outsources” functionality to other classes by aggregation of multiple specialized classes. This is similar in spirit to extension, but does not attempt to “philosophically” integrate the functionality under the umbrella of the same class.

The choice of inheritance and delegation is a policy choice, often motivated by a heuristic philosophy. The Liskov Substitution Principle (Class 2 “IS A” Class 1) has often been used to explain when inheritance should be used over delegation, for instance[22]. However, the “is a” relationship is a heuristic one, in which it is unclear whether one is talking about syntax or behaviour. This often leads to ambiguities in its application.

Inheritance is thus used with several inequivalent meanings in OO, and so we must disentangle the semantics of these distinct policies as behavioural promises.

8.3.1 Extension

Extension is a relationship that decides whether a parent class is a subset of the derived child class. Let us define these in terms of promises.

Definition 15 (Extension) *Let c and c' be two class-like bundles, We say that c extends c' or $c > c'$ iff $c' \subseteq c$. This gives us the notion of a container, and containment.*

Notice that this is simply a rule for aggregation. In terms of promises, there is no difference between the extension of a class and the formation of a bundle.

8.3.2 XOR Overriding

Overriding promises (methods and values) is typically a technique used in sub-type polymorphism. When overriding, there cannot be a conflict between parent and client by definition, since the new behaviour and the old behaviour are mutually exclusive. Overriding requires a switch-like structure (indeed subtype-polymorphism is often intended to replace switch-case constructions). Let S and R be agents, promise bundles can be selected and hence overridden based on type labels using a construction in which bundles are made exclusively if a certain type is promised. Since types are mutually exclusive, the promises are mutually exclusive:

$$\begin{array}{ccc}
S & \xRightarrow{\Pi:c_1/\text{type}=\text{type1}} & R \\
S & \xRightarrow{\Pi:c_2/\text{type}=\text{type2}} & R \\
R & \xrightarrow{\pi:\text{type}} & S \\
S & \xrightarrow{\pi:U(\text{type})} & R
\end{array} \tag{28}$$

8.3.3 Specialization

Specialization is a selective use of overriding to alter a subset of promises.

Definition 16 (Specialization) *Let c_p and c_i , $i = 1 \dots N$ be class-like bundles. We say that*

- c_i specializes c_p

iff a subset $c \subseteq c_p$ of class promises in the bundle is made conditionally on predicate p and all promises in c_i are dependent on predicates p_i , in such a way that p and p_i are all mutually exclusive. It is normally assumed that the promise types in c_p and c_i are identical.

Thus we define a polymorphic class replacement using a simple switch-case construction as one would expect.

For example, consider promise bundles that belong to the parent and the child. These are conditionally switched if the promise of a subtype is made by the receiving agent.

$$\begin{array}{ccc}
S & \xRightarrow{\Pi:c_{\text{base-parent}}} & R \\
S & \xRightarrow{\Pi:c_{\text{ext-parent}}/\neg\text{subtype}} & R \text{ (optional)} \\
S & \xRightarrow{\Pi:c_{\text{child}}/\text{subtype}} & R \\
R & \xRightarrow{\pi:\text{subtype}} & S \\
S & \xRightarrow{\pi:U(\text{subtype})} & R
\end{array} \tag{29}$$

Here the extension-parent class bundle is exchanged for the child bundle if subtype is promised. The base-parent bundle is promised in both cases. If a parent class wishes to enforce behavioural constraints on a child, the it must make such base-promises that are not overridable, otherwise there can never be any broken promises in spite of radically changing behaviour.

Currently there is no way of having behavioural promises encoded always in class implementations in programming languages as far as the authors know.

8.3.4 Substitution

Specialization is a selective use of overriding to alter a subset of promises.

Definition 17 (Substitution) Let c_p and c_i , $i = 1 \dots N$ be class-like bundles, We say that

- c_i substitutes c_p

iff the body c_p in the bundle is made conditionally on some predicate p and all promises in c_i are dependent on predicates p_i , in such a way that p and p_i are all mutually exclusive. It is assumed that c_p and c_i are isomorphic with respect to promise types and number, so that every c_i is a complete replacement for c_p .

8.3.5 The Liskov test

Our notion of substitution is stronger than the weak form often used in the literature. The Liskov substitution test[22] attempted to summarize and make sense of the practice of inheritance by introducing a heuristic test of compliance. This is often described in terms of the informal “is a” relation, whose name seems to imply an equality of certain entities, but this is not the case. We make sense of this relation with our own definition as follows:

Definition 18 (“is a”) *A class-like promise bundle $S \xrightarrow{\Pi:c} R$ “IS A” class-like promise bundle $S \xrightarrow{\Pi:c'} R$ iff both bundles can be promised at the same time without breaking any promises.*

A common example used to illustrate this is that of a rectangle and a square. Clearly a square is a special case of a rectangle, in every mathematical sense, however a square has fewer behavioural freedoms than a rectangle (not more) so it cannot extend the notion of rectangle. Informally it seems that square is a rectangle, but this is because we have used a generic classification; the difficulties arise when we try to specify.

Let us examine the promises made by a square and a rectangle and see how the behavioural (promise) definition above indicates why a square “not is a” rectangle, according to our definition.

Let R be a rectangle agent, S be a square agent and A be any other promisee agent.

$$\begin{aligned}
 R & \xrightarrow{\pi:\text{width}=w} A \\
 R & \xrightarrow{\pi:\text{height}=h} A \\
 R & \xrightarrow{\pi:\text{angle}=90} A \\
 R & \xrightarrow{\pi:\text{sides}=4} A
 \end{aligned} \tag{30}$$

We assume they have a common data-type schema, so we let square extend rectangle:

$$\begin{aligned}
 S & \xrightarrow{\pi:\text{width}=w} A \\
 S & \xrightarrow{\pi:\text{height}=h} A \\
 S & \xrightarrow{\pi:\text{angle}=90} A \\
 S & \xrightarrow{\pi:\text{sides}=4} A \\
 S & \xrightarrow{\pi:w=h} A
 \end{aligned} \tag{31}$$

The only difference now is that an additional promise has been made. This is a behavioural constraint (not easily implementable in a programming language,

but straightforward in terms of promises). Now, since the promises

$$\begin{array}{l} S \xrightarrow{\pi:\text{width}=w} A \\ S \xrightarrow{\pi:\text{height}=h} A \\ S \xrightarrow{\pi:w=h} A \end{array} \quad (32)$$

are not independent, they may be reduced to:

$$\begin{array}{l} S \xrightarrow{\pi:\text{width}=h} A \\ S \xrightarrow{\pi:\text{height}=h} A. \end{array} \quad (33)$$

And if we now try to ask the question whether square “is a” rectangle, it implies that all promises of S and R must hold simultaneously. However, we quickly find a broken promise:

$$\begin{array}{l} S/R \xrightarrow{\pi:\text{width}=w} R \\ S/R \xrightarrow{\pi:\text{height}=h} R. \end{array} \quad (34)$$

Hence a square “is not a” rectangle in terms of the behaviours we have attributed to it. Why was this problem detectable here and not in a classical programming inheritance regime? Clearly the answer is that promises can be made about behaviour, but normal program syntax does not represent behavioural constraints. To avoid this problem one can therefore introduce the policy, “Parent bundle promises should never be broken by a child bundle”.

8.4 Other class structures

The conceptual problems that occur with OO inheritance are only partially illuminated by the Liskov substitution principle. The idea of inheritance is itself unclear because it derives only loosely from a heuristic model of phylogentic attributes (family trees). We want to map types to behaviours (this is what promise theory does). What OO does is map types to aggregates without a language to define behavioural constraints. “Is a” is not a good metaphor for inheritance, since it is not obviously behaviour constraint. Our interpretation of “is a” above provides one explanation for inheritance, but one can easily question whether the name and the definition match.

The limitation that inheritance is a purely tree-like structure is artificial (even in biology, where bacteria exchange genetic attributes without spawning a

third party). The Service Oriented Architecture is somewhat freer in this respect. An SOA application allows something that is not allowed in OO, is the following scenario.

Example 4 *Company Yin produces a system that allows third-party add-ons by company Yang. Yin leaves hooks into its code available as heritable classes to the third party Yang and asks for it to do the same. In this way, both can use each others default code or customize it to give each other the maximum advantage to their business. For example, specially extended device drivers that make proprietary extensions. What they need to do is this:*

```
class yin : inherits yang
{
// yin stuff
}

class yang : inherits yin
{
// yang stuff
}
```

This scenario is strictly forbidden by OO languages. The two companies could spawn a child and, using dual-inheritance, allow it to inherit from both parents. In terms of promises, there is no logical problem with this, as long as there are no namespace conflicts. Indeed this symmetric stand-off, according to a different philosophy, is the basic economic model of a game (trade). It is explicitly disallowed in OO programming. In the Service Oriented Architecture it is possible.

We are not advocating this necessarily as a weakness of OO; some readers will no doubt think the example is simply ridiculous, because their world view is one that accepts the OO way of thinking. Rather we propose that the weakness in any programming philosophy lies in believing that features are “more easily understood by humans” if they are presented through analogies. As long as languages offer mechanisms that are cluttered with conceptual baggage, there will be a need for meta-policies or design patterns to solve specific scenarios.

9 A natural modelling example

9.1 A bank account scenario

An example scenario that has been shown to be problematic in terms of Unified Modelling Language (UML) is the bank account scenario described in ref.

[23]. Consider a bank, comprising a number of employees and accounts for customers. The issue here is that bank customers should have access to use their accounts and that bank employees can make certain privileged transactions on accounts. However, if the customers are in fact bank employees, they should not have privileged access to perform transactions that include their own accounts (since this could lead to embezzlement). Any user/Person, whether the owner of an account or not should be able to make a cash payment to any account. This example is somewhat simplistic, but therefore also uncomplicated; however, it illustrates the basic conundrum.

A UML model of this scenario is shown in fig. 4. We introduce the parent class “Person” from which two child classes are derived by inheritance. Unlike promise theory, UML seeks to model *associations* between classes; in UML a subclass inherits associations, as these form a subset of general attributes. Given only this guidance, one could easily be led to the model shown in fig. 4. This structure has two problems that are easily avoidable by reorganizing the

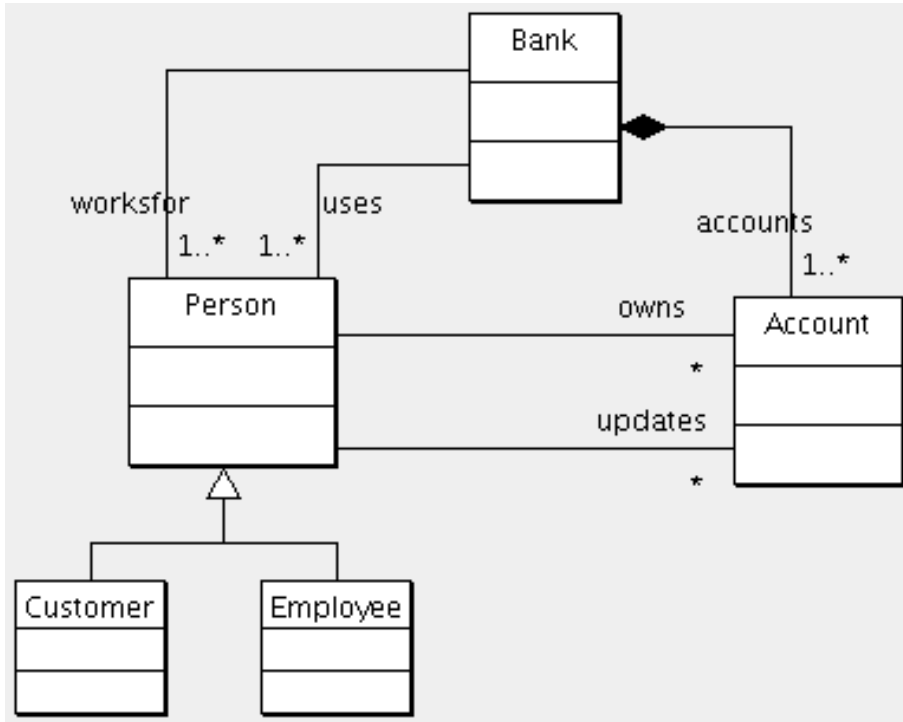


Fig. 4. An OO class model for the bank account/user interaction, in UML.

model, but which are not ruled out by UML:

- (1) It assumes that customers and employees are mutually exclusive categories.
- (2) It allows employees to perform privileged updates on their own accounts.

Although these are undesirable properties of the erroneous model, UML semantics do not offer sufficient guidance, without additional policies, to see

what is going wrong. In this example, we would like to show that the basic principles of promise theory (no agent can be forced, and knowledge is *a priori* private) lead us quickly to a cleaner and more intuitive structure which obviates the need for an explicit “ownership” relation altogether.

We introduce agents for users, bank accounts and also a single (optional) agent representing the bank’s administration. Always bearing in mind that agents cannot be forced to perform any function, and have limited knowledge, we identify the necessary promises from Users to Account agents. We further introduce a category attribute for users of accounts: “customer”, “employee” and the default “other”. The categories “customer” and “other” are disjoint.

We assume one account per agent and one user/Person per agent. To transport the necessary information to make promise-theoretic decisions, a user/Person-agent must identify itself and its category. The *possible* set of promises that are common to all agents are:

$$\text{User/Person} \xrightarrow{\pi:\text{name=identity}} \text{Account} \quad (\text{general}) \quad (35)$$

$$\text{User/Person} \xrightarrow{\pi:\text{cash payment}} \text{Account} \quad (\text{general}) \quad (36)$$

$$\text{User/Person} \xrightarrow{\pi:\text{customer}} \text{Account} \quad (\text{subset}) \quad (37)$$

$$\text{User/Person} \xrightarrow{\pi:\text{employee}} \text{Account} \quad (\text{subset}) \quad (38)$$

Some promises probably only make sense for certain categories of agent and we might represent these as conditional promises:

$$\text{User/Person} \xrightarrow{\pi:\text{use account/customer}} \text{Account} \quad (39)$$

$$\text{User/Person} \xrightarrow{\pi:\text{priv update/employee}} \text{Account} \quad (40)$$

however, one could imagine that dishonest agents might try to make these promises unconditionally. It is entirely up to the receiving account to deal with such behaviour, according to the promise precepts, so we shall disregard the conditionals here, as they make an unwarranted assumption about agent behaviour. Note that an agent which does not promise to be a customer or an employee can still exist and make promises, such as cash payments to any account. Privileged updates, on the other hand, should only be promised by employees.

An account agent makes a number of promises to users in return; some of these are of a general nature, such as accepting information promised to the agent:

$$\text{Account} \xrightarrow{\pi:U(\text{name=identity})} \text{User/Person} \quad (41)$$

$$\text{Account} \xrightarrow{\pi:U(\text{employee})} \text{User/Person} \quad (42)$$

$$\text{Account} \xrightarrow{\pi:U(\text{customer})} \text{User/Person} \quad (43)$$

$$\text{Account} \xrightarrow{\pi:\text{Keep money safe}} \text{User/Person} \quad (44)$$

$$\text{Account} \xrightarrow{\pi:\text{Account functions}} \text{User/Person} \quad (45)$$

$$\text{Account} \xrightarrow{\pi:U(\text{cash payment})} \text{User/Person} \quad (46)$$

Some additional promises are conditional on the category of user, such as the promise to accept instructions from different user categories. This leads us naturally to an implementation in terms of access control rather than class inheritance.

$$\text{Account} \xrightarrow{\pi:U(\text{use account})/C_1} \text{User/Person} \quad (47)$$

$$\text{Account} \xrightarrow{\pi:U(\text{priv update})/C_2} \text{User/Person} \quad (48)$$

Let us consider these conditions. Any agent can make a cash payment to an account, regardless of privilege level. For other transactions, we want to ensure authorization however. The owner of an account should have all normal usage privileges, thus if an agent with a name matching. The conditions C_1, C_2 must be based in information that is available (promised) to the account agents:

$$C_1 : \text{name} = \text{owner} \quad \text{AND} \quad \text{employee} \neq \text{true} \quad (49)$$

$$C_2 : \text{name} \neq \text{owner} \quad \text{AND} \quad \text{employee} = \text{true} \quad (50)$$

These conditional promises are mutually exclusive and hence they should be represented by exclusive promises, which *extend* the unconditional promises to accept cash payments.

We allow, by contrast with the UML model, user/Person agents to be both customers and employees non-exclusively, since the conditional tests can be made based entirely on the information that a user is operating in employee mode (e.g. making that promise by entering an authorization code or passwd). We see from the services provided by an account that users are distinguished entirely by the functions they carry out, not by named skills without a basis in service. Thus it is only allowed behaviour that distinguishes user categories.

Account-agents promise to accept normal usage transactions as long as an agent is not an employee (any agent customer or not might need to pay money into a customer's account). An agent who is *sometimes an employee* can also use its account as long as it is not currently making the employee promise. As soon as employee privilege is invoked, account agents invoke different promised behaviour which restricts the actions they can perform.

What about the UML ownership association? The account attribute “owner” is the private knowledge of the account-agent concerned. There is no need for this information to be issued as an explicit relation that is promised to a user, since that behaviour is implicit in the Use-promises. Thus the “owns” association is not a necessary interface relation, only an attribute of the account.

Readers might be uncomfortable that a bank account somehow knows ownership information by itself – this is perhaps not how many would view a traditional bank. If we wish to model a more traditional, centralized bank, we could introduce a central bank agent which has knowledge of all accounts and delegates the data to its subordinate account-agents:

$$\text{Bank} \xrightarrow{\pi:Name_i} \text{Account}_i, \forall i \quad (51)$$

Similarly, the account-agents could place themselves under the umbrella of the bank by promising to use these names from the main agent.

$$\text{Account}_i \xrightarrow{\pi:U(Name_i)} \text{Bank} \quad (52)$$

$$\text{Account}_i \xrightarrow{\pi:K_{name}=Name_i} \text{Bank} \quad (53)$$

There is nothing in this model that requires this however. If one were to implement a bank as a Service Oriented device, then bank accounts could be created anywhere without a common umbrella. They do not have to belong to a single bank. Our model is rather too primitive to make a detailed case for this here however.

9.2 The roles and classes

In promise theory, we have conjectured that the natural spanning set of classes in a system can be discovered by looking for *roles* in the promise graph, i.e. common and repeated patterns within the graph. When role-promises are disjoint or mutually exclusive they fall naturally into separate sub-classes, such as might be implemented through sub-type polymorphism in OO. The solution below is shown in fig. 5.

We recognize several kinds of user or person promise roles: promises made by all users, and those which are made only by employees or customers. As we pointed out above, the promises made only by customers and employees are exclusive only as a courtesy and we can easily do away with the conditional aspect of those promises. We choose that approach here for parsimony. Hence we have a role for agents who promise to be customers, a role for employees and a role for others (making neither of the other promises above). There is a

role for those who promise both. Since the former roles overlap, it is incorrect to represent these as subtypes and we propose that a person/user class would be of the form of a single data-type.

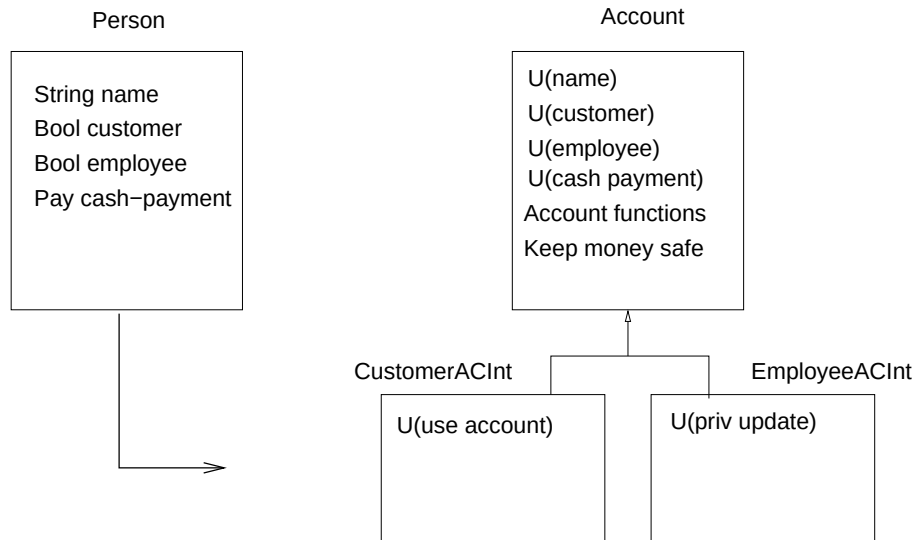


Fig. 5. A promise role derived class structure for the bank account problem. Note that the customer/employee roles are not represented as separate classes, but as attributes to the common user (Person) class.

We recognize three kinds of account promise roles, two of which are mutually exclusive (and hence form natural subtypes) and one which is common to the others (and hence requires no subtype). This forms a natural tree decomposition as shown in fig. 5.

What we see is that the promise theory solution, which makes behaviour the premier consideration, is to split the account class into two rather than the person/user class. The split is based on what promises are available to the different kinds of users. The change to an account subtype is a change in security level (like a pure Clark-Wilson model[24] interface) which leads to a natural access control. The decomposition is not based on names (as in the OO philosophy) but on the promised behavioural characteristics. This shows us immediately how to build such a system as a distributed Service Oriented system, but it is equally applicable to a strategy of “need to know” in a local system.

We end up with two security levels, implemented through mutually exclusive interfaces (modes) in the account class, rather than two kinds of users, since this arrangement embodies the behavioural constraints.

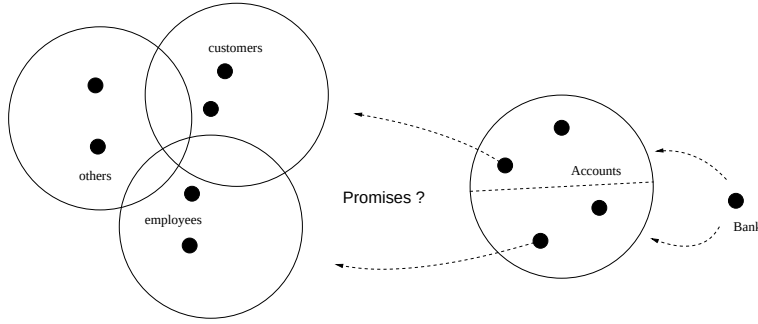


Fig. 6. Taking a promise viewpoint, the roles in the bank account interaction example are shown. Note that different user roles overlap. An employee might also be a customer in the bank, similarly user can pay in transactions even when a user is not a customer of the bank.

10 Discussion and conclusion

Using promises, we are able to model some salient features of both the Object Oriented paradigm and the Service Oriented Architecture in a common framework. There is an enormous potential for describing interfaces and constraints on program activities, without having to deal with actual computational processes. *Design patterns* are explicit embrace of policy, within policy – to find standard ways of using the standard tools. They are indications of the incompleteness of programming paradigm policies such as OO. Incompleteness is not a bad thing – it means flexibility, but it indicates that the doctrine of ‘better programming’ implied by some authors warrants further scrutiny.

Another programming paradigm introduced recently is that of *aspects*. Aspects allow one to weave in-line code fragments into an existing program using a secondary compiler. The break with the hierarchical container idiom, but are easily represented as promises or services performed by a weaving agent.

A practical benefit of our promise theory approach in OO class construction is the following. In OO design one models data-types first and creates instances of these pre-decided types on demand as program logic develops. This is not unnatural, since one does not necessarily know what precise objects will be required in advance; however, it can lead to blind alleys and refactoring, as one has no empirical basis for a project at the outset. A spanning set model, in terms of actual objects and their promises, which leads to ‘natural’ organizational structures, can help to avoid some of these blind alleys. We have applied this approach to code examples in cfengine 3[25] with some success (see fig. 7) as a proof of concept.

Compilers are inadequate to find many problems in programs and programming design policy because they check syntax not behavioural promises. If programming languages were able to promise behaviour, compilers would also

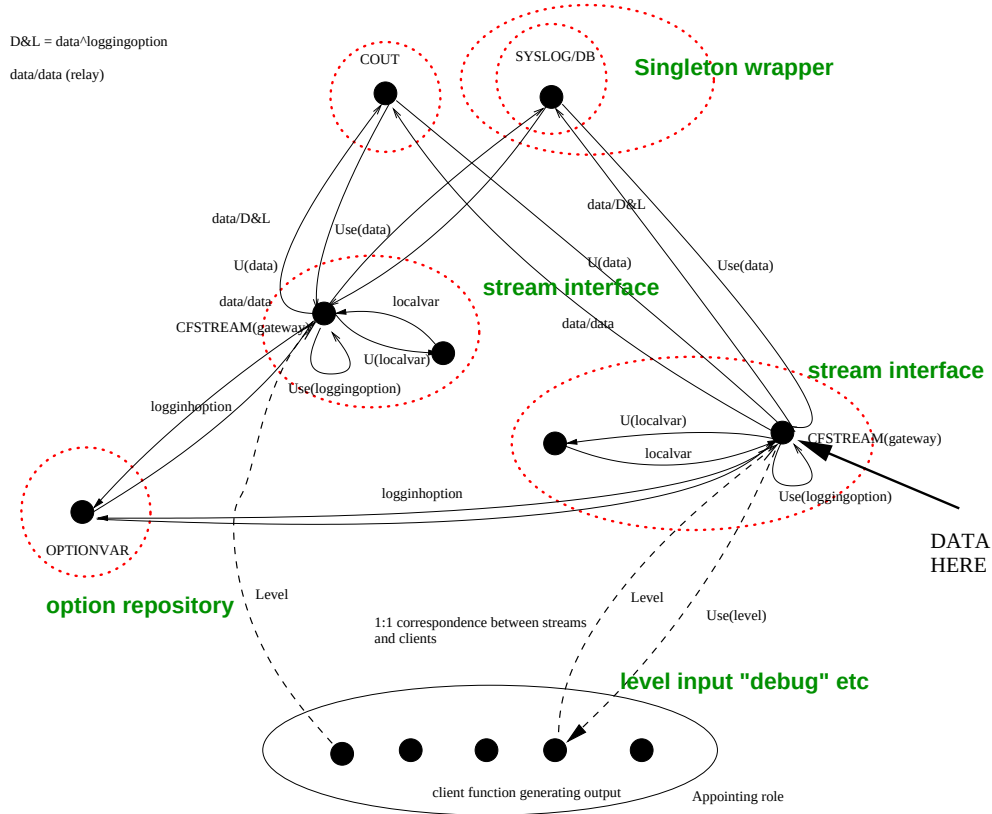


Fig. 7. Modelling the design of logging and output stream service using promises. The dotted ellipses show how natural roles emerge which can be identified with separate class type-bundles

be able to detect improper use of policies like inheritance etc.

Promises also allow analyses, both logical and economic[7,8]. The economic trade-like model inherent in the SOA is naturally described in promise theory[8]. Using the idea of common knowledge as a graph theoretic construction (rather than a modal logic inference) one has an easy way of tracking scope with graph theoretical tools. This could be an approach to examining *reflections*, or ‘self-aware’ programs.

There are other ways entirely in which promises might assist in modelling software engineering. The act of basing software on a specification is a promise to customers, saying that a program will comply with its specification. On the other hand, much software is developed without any formal specification. Open source software, for example, is often the result of dedicated work by ‘hackers’ who based their work more on knowledgeable intuition than on formal methods. Does this make it less able to function reliably? Users clearly use the software even though the software promises nothing formally. We hope to return to some of these issues in future work.

Acknowledgement: we are grateful to Jan Bergstra, Alva Couch and Tuva Hassel Stang for stimulating discussions.

References

- [1] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997.
- [2] Roger S. Pressman, editor. *Software Engineering: a Practitioner's Approach*. McGraw-Hill Higher Education, Boston, 5th edition, 2004.
- [3] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.
- [4] Olaf Zimmermann, Pal Krogdahl, and Clive Gee. Elements of service-oriented analysis and design. In *IBM DeveloperWorks article*. 2004.
- [5] A. Alexandrescu. *Modern C++ design*. Addison Wesley, 2001.
- [6] Mark Burgess. An approach to understanding policy based on autonomy and voluntary cooperation. In *IFIP/IEEE 16th international workshop on distributed systems operations and management (DSOM), in LNCS 3775*, pages 97–108, 2005.
- [7] M. Burgess and S. Fagernes. Pervasive computing management: A model of network policy with local autonomy. *IEEE Transactions on Software Engineering*, page (submitted).
- [8] M. Burgess and S. Fagernes. Voluntary economic cooperation in policy based management. *IEEE Transactions on Software Engineering*, page (submitted).
- [9] J. Strassner. *Handbook of Network and System Administration*, chapter Knowledge Engineering Using Ontologies. Elsevier Handbook, 2007.
- [10] J. Vranken and K. Koymans. *Handbook of Network and System Administration*, chapter Application Integration using Semantic Web Services. Elsevier Handbook, 2007.
- [11] J.A. Bergstra and A. Ponse. Interface groups for analytical execution architecture. (*preprint*), 2006.
- [12] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proc. of the 3rd IJCAI*, pages 235–245, Stanford, MA, 1973.
- [13] M. Wooldridge. *An Introduction to MultiAgent Systems*. Wiley, Chichester, 2002.
- [14] J. Ferber. *Multi-agent Systems: Introduction to Distributed Artificial Intelligence*. Addison Wesley, 1999.

- [15] M. Burgess and S. Fagernes. Pervasive computing management: Applied promise theory. *Proceedings of the 1st IEEE International Workshop on Modelling Autonomic Communications Environments (MACE)*, page (in press), 2006.
- [16] M. Burgess and K. Begnum. Voluntary cooperation in a pervasive computing environment. *Proceedings of the Nineteenth Systems Administration Conference (LISA XIX) (USENIX Association: Berkeley, CA)*, page 143, 2005.
- [17] W3C. Defining n-ary relations on the semantic web: use with individuals. <http://www.w3.org/TR/2004/WD-swbp-n-aryRelations-20040721/>.
- [18] E.L. Robertson. Triadic relations: An algebra for the semantic web. *Lecture Notes in Computer Science*, 3372:91–108, 2005.
- [19] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 21, pages 38–45, New York, NY, 1986. ACM Press.
- [20] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 21, pages 214–223, New York, NY, 1986. ACM Press.
- [21] William R. Cook. A denotational semantics of inheritance. Technical Report CS-89-33, 1989.
- [22] B. Liskov and J. Wing. FAMILY VALUES: A BEHAVIORAL NOTION OF SUBTYPING. Technical Report MIT/LCS/TR-562b, 1993.
- [23] Demissie B. Aredo. *Formal Development of Open Distributed Systems: Integration of UML and PVS*. PhD thesis, Department of Informatics, University of Oslo, Norway, 2004.
- [24] D.D. Clark and D.R. Wilson. A comparison of commercial and military computer security policies. *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, page 184, 1987.
- [25] M. Burgess. Cfengine www site. <http://www.cfengine.org>, 1993.