Commitment-Based SOA[*]

Munindar P. Singh, Amit K. Chopra, and Nirmit Desai

{singh, akchopra, nvdesai}@ncsu.edu

Abstract

The vision of service-oriented computing is centered on business services. By contrast, existing service-oriented architectures are formulated in terms of low-level abstractions far removed from business services. We describe a new architecture whose components are business services and whose connectors are patterns (modeled in terms of commitments) that support key elements of service engagements. We contrast this architecture with existing SOAs.

Introduction

The services vision promises the creation of a dynamic web of value. According to this vision, anyone with something of value to offer can easily create and deploy a corresponding service; anyone needing to benefit from that value need simply select such a service and incorporate it into a desired solution. In other words, services are software components created, deployed, and flexibly composed to yield desired applications—or more services.

Current SOAs purport to support the services vision. But there is a fundamental discrepancy between the vision and its realization in current SOAs. When the services vision is promulgated—and the reason many find it compelling—the implication is that the services are business services. However, current SOAs interpret services far more narrowly—as surrogates for (computational) objects. Whereas business services are *engaged* (often involving subtle business considerations), objects are *invoked* (with business considerations hidden within computational artifacts). More importantly, business services are usually autonomous entities that come together in a service engagement.

Let's consider the familiar purchase scenario as modeled in the leading SOA approaches. The purchase (of, say, books) is a business service, which combines services such as ordering a book, paying, and shipping. Any or all of these services could be provided by different organizations. Both BPMN[1] (Business Process Modeling Notation) and BPEL[2] (Business Process Execution Language) represent composed services as processes specified via control and data flows over tasks (the differences[3] between them are syntactic). For example, BPMN would model purchase as three tasks—ordering, paying, shipping—where control and data (book identifier and price) flow from ordering to both paying and shipping. WS-CDL[4] (Choreography Description Language), another

[*]This research was partially supported by the US National Science Foundation under grant DST-0139037.

[1]http://bpmn.org

[2]http://docs.oasis-open.org/wsbpel/2.0

[3]http://bpmn.org/Documents/Mapping%20BPMN%20to%20BPEL%20Example.pdf

[4]http://www.w3.org/TR/ws-cdl-10

leading approach, specifies how services exchange messages. WS-CDL would specify how the ordering service sends messages to both the paying and shipping services, which perform their work upon receipt of such messages. Declarative approaches for constraining task or message order and occurrence improve over the above with respect to modularity and inspectability [1, 2], but continue to emphasize control and data flow.

Commitments and CSOA

In contrast with existing approaches, our commitment-based SOA (CSOA) gives primacy to the *business meanings* of service engagements, which are captured through the participants' *commitments* to one another. CSOA constrains tasks or messages only where doing so affects the business meaning. Computationally, each participant is modeled as an *agent*; interacting agents carry out a service engagement by creating and manipulating commitments to one another.

A commitment relates three parties: a *debtor* who is committed to a *creditor* typically within the scope of an organizational *context*. The context may be an institution—for example, a marketplace (such as eBay) or a legal jurisdiction (such as California)—in which the interaction occurs. Members of an institution who fail to *discharge* their commitments risk sanction. UCC[5] (Uniform Commercial Code) applies in many US jurisdictions, and dictates conditions such as when a customer need not pay for goods he bought (for instance, if the goods arrive damaged and the customer returns them immediately). In general, the context is crucial in handling exceptions, which are rife in business settings. For modeling purposes, we treat the context as an agent in its own right.

Importantly, commitments can be manipulated, which supports flexibility. A debtor may *create* a commitment thus activating it, or *discharge* it thus satisfying it. Given a commitment, its creditor may *assign* it (to a new creditor) and its debtor may *delegate* it (to a new debtor). A debtor may *cancel* a commitment, whereas a creditor may *release* the debtor from the commitment.

Consequently, CSOA offers the following benefits:

Enactment and Compliance Service enactments can be judged correct as long as the parties don't violate their commitments. This enhances flexibility over traditional approaches by expanding the operational choices for each party [3]. For example, if the customer substitutes a new way to make a payment or elects to pay first, no harm is done, because the behavior is correct at the business level. And, the seller may employ a new shipper; the buyer may return damaged goods for credit; and so on. Conversely, a customer would be in violation if he keeps the goods but fails to pay. In this manner, commitments support business-level compliance and don't dictate specific operationalizations [4]. By contrast, without business meaning, exercising any such flexibility would result in noncompliant executions.

Specification and Composition Commitment-based specifications can explicitly reflect requirements at the business level, which is natural for stakeholders. For example, upon placing an order, the customer becomes conditionally committed to the merchant to paying for the goods if they are delivered. The delivery of the goods commits the customer to paying for them. When the customer pays, this commitment is discharged. Commitments provide clean conceptual boundaries at which we can compose engagements. For example, we can reuse

[5]http://www.law.cornell.edu/ucc

order placement in specifying a more complex service engagement in which independent delivery and payment services are employed. By contrast, without business meaning, we would lack a basis to establish the validity of any reuse or composition.

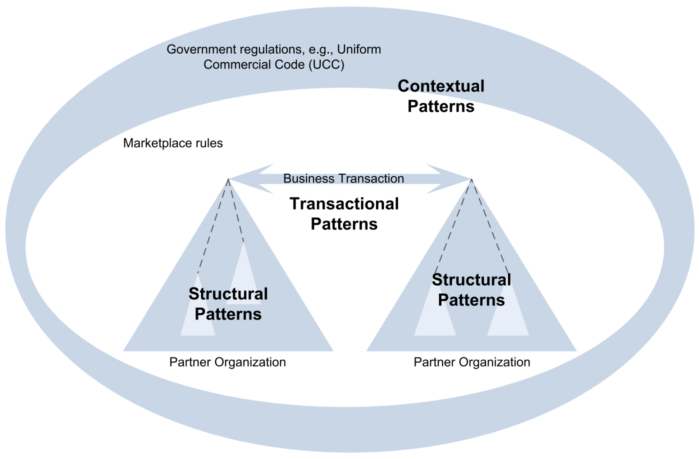


Figure 1: Classifying CSOA patterns

CSOA is characterized by a family of reusable *patterns* that arise in a variety of service engagements. Figure 1 shows three elements of a service engagement, which help classify our patterns. *Transactional* refers to the dealings among two or more participants. *Structural* refers to how a participant (including subcontractors) is organized. *Contextual* refers to the organizational context in which the engagement takes place. A typical service engagement model would include several patterns, each applied to some participants and their commitments.

What follows are descriptions of illustrative patterns of each category. These patterns are induced from existing processes including UCC, RosettaNet[6], TWIST[7] (Transaction Workflow Innovation Standards Team), the MIT Process Handbook (MITPH)[8], and extended transaction models [5].

Commitments as a Basis for Patterns

The formula $C(debtor, creditor, context, precondition, condition)$ means that $debtor$ commits to $creditor$ in $context$ that if $precondition$ becomes true, the debtor would bring about $condition$. When $precondition$ holds, the commitment undergoes a *detach*, meaning that the debtor becomes unconditionally committed to bringing about $condition$.

Background: Statecharts

Figures 2–4 depict patterns as statecharts [6], which are more expressive than finite state machines. Let's review Figure 2(a) to illustrate our notation. Rounded rectangles with labels denote states.

[6]http://www.rosettanet.org
[7]http://www.twiststandards.org
[8]http://ccs.mit.edu/ph

States may contain other states. For example, null and active (containing conditional and base) are states. Arrows labeled with events denote transitions between states. Transitions without labels are always enabled. An arrow from a containing state denotes a transition from *each* of its substates (for example, *discharge*).

Figure 2(a) captures the lifecycle of a commitment: null means it does not exist, active means it is fully in force, satisfied means it has been discharged, and violated means it cannot be discharged. A commitment in base may become violated, whereas a commitment in conditional cannot directly become violated, but goes to null on expiration. For example, a customer may offer to buy some goods by creating the commitment: "if you ship I will pay." The commitment may expire or the customer may pay. If the merchant delivers, that would *detach* the commitment, unconditionally committing the customer to pay.

We underline commitment labels in the patterns: the big rectangle labeled commitment is a state that contains the other states. This reflects the system being in a state where commitment is in force.

Importantly, *create* is performed explicitly whereas *detach* and *discharge* are caused automatically when *precondition* and *condition* respectively hold; *expire* is caused implicitly upon timeout; but *cancel* may be performed explicitly or occur via timeout.

An active commitment must be either in state conditional or base, and this depends solely on whether its precondition holds (base) or not (conditional).

Most patterns involve more than one commitment. Such patterns are described via a *composite* state that has a concurrent component for each commitment. For example, in Figure 2(c), a dotted vertical line separates original and new. A composite state can be thought of as a list of its components, which in logical terms are *AND*ed together. Thus, if original is in state null and new is in state active, the composite state is given by original being in null *AND* new being in active.

A solid bar with incoming and outgoing arrows is a synchronization primitive: when all events corresponding to the incoming arrows happen, the transitions corresponding to each outgoing arrow also execute. (Throughout, SMALL CAPS indicate comments.)

Pattern Language

Of the 13 attributes in the classical template for object-oriented design patterns [7], the following attributes are relevant for commitment-based patterns: *name and classification*; *intent*; *motivation*; *applicability*; *consequences*; *implementation*; *known uses*. A common consequence is that the parties involved be proactive and able to communicate flexibly: this is why we model them as agents.

Engagements at the level of business entities is what each pattern is about. The *implementation*, specified via a statechart, incorporates the *participants* and *structure*. To make the patterns modular, we include in its statechart only the relevant states and transitions for each commitment involved. (In this sense, our statecharts are not individually complete, and rely upon other patterns to have brought about the states from which they begin.) The commitment operations corresponding to a transition would be realized via business actions such as sending purchase orders, delivering goods, and so on, thus enacting the corresponding business scenarios.

Transactional Patterns

The core of a service engagement concerns the business transaction that it seeks to accomplish. Transactional patterns describe the corresponding interactions in terms of how the associated commitments are created and manipulated. These patterns deal with common transactional primitives

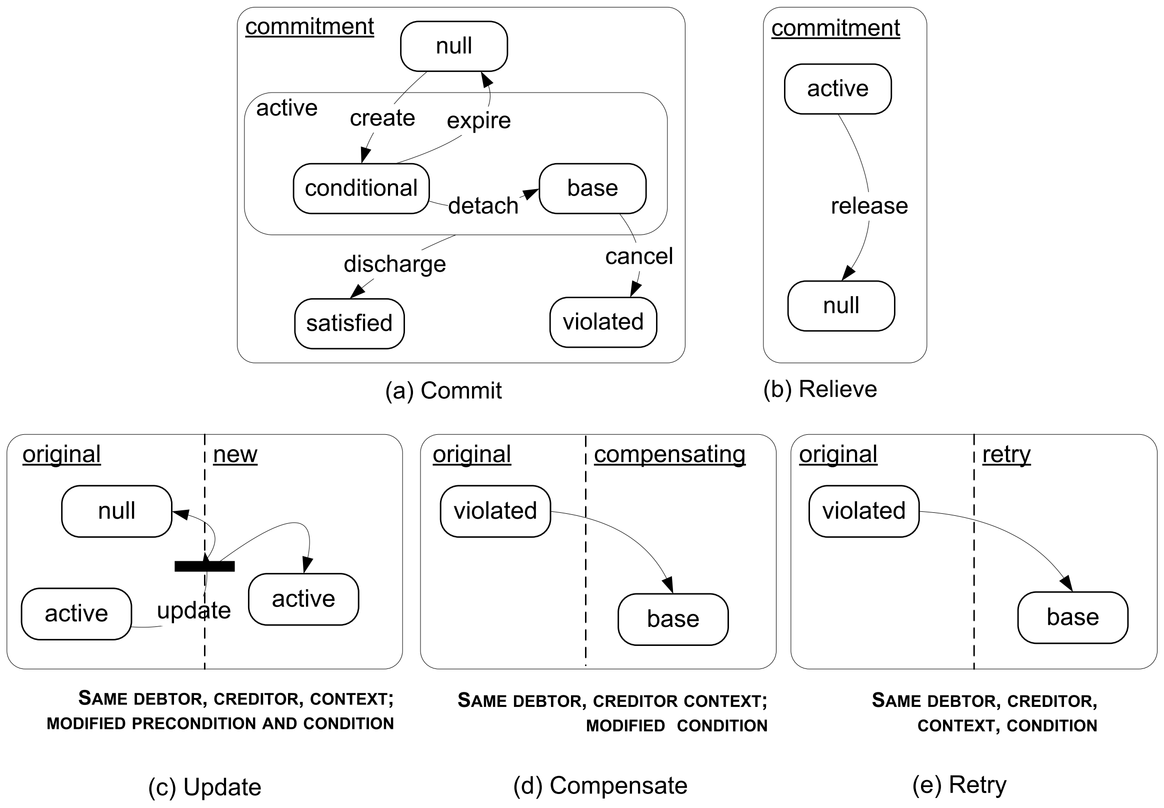


Figure 2: Transactional patterns

such as initiating a business transaction, formally creating suitable commitments, satisfying the commitments, and possibly updating, retrying, or compensating actions in light of stated gating conditions. Each of these patterns involves the same two participants.

Commit
Intent: Expressing an offer
Motivation: Explained in the text
Applicability: Whenever an offer is made as part of setting up a service engagement
Consequences: For progress, the creditor should be ready to bring about the precondition
Implementation: Figure 2(a)
Known uses: Purchase, MITPH purchase, RosettaNet purchase order (PIP3A4)

Relieve
Intent: The creditor releases the debtor from the commitment
Motivation: If the customer obtains the book he needs before the merchant has shipped it, he may release the merchant from the commitment to ship
Applicability: This reduces cleanup work for both customer and merchant, and helps maintain good business relations
Consequences:
Implementation: Figure 2(b). Upon *release*, commitment becomes null
Known uses: Purchase, MITPH notify, RosettaNet purchase order cancel (PIP3A9)

Update
Intent: A commitment is updated
Motivation: An update is needed when the customer asks for a hardcover book instead of a paperback; the original commitment is nullified
Applicability: A participant's needs might change while a long-lived engagement is in progress
Consequences:
Implementation: Figure 2(c). Upon *update*, original becomes null, and new is created
Known uses: Purchase, MITPH update, RosettaNet purchase order change (PIP3A8)

Compensate
Intent: Some business action needs to be undone
Motivation: A customer sends payment, which commits the merchant to sending the goods; later, if the merchant fails to deliver the goods on time thus violating its commitments, it needs to make amends, for example, by refunding the payment
Applicability: Supporting an extended form of transactional rollback, so as to maintain an all or none effect despite exceptions [5]
Consequences: Typical usage is when the debtor is unable to bring about the condition of original
Implementation: Figure 2(d). Upon violation of original, creation of compensating is enforced
Known uses: Purchase, RosettaNet return product (PIP3C1)

Retry
Intent: Some business action needs to be redone
Motivation: When a shipment doesn't arrive, the merchant resends it
Applicability: Supporting an extended form of transactional rollforward, so as to maintain an all or none effect despite exceptions [5]
Consequences: Typical usage is when the debtor made an attempt to bring about the condition of original, but the effort proved unsatisfactory
Implementation: Figure 2(e). Upon violation of original, creation of retry is enforced
Known uses: Purchase, MITPH rework

Structural Patterns

Service engagements involve subtle relationships among the parties involved. The structural patterns capture such relationships in terms that are not directly involved in a transaction. Specifically, the structural patterns may capture constraints on which party may play which role, or whether a party may delegate or assign certain commitments to another party. Each of these patterns involves two or more participants, all of who are debtors (delegators or delegatees) of commitments. For brevity, we elide the dual cases of creditors assigning commitments.

Structural patterns can be most simply illustrated when a service engagement involves an organization with internal structure. A participating organization may delegate its commitments under the engagement to appropriate members (who could themselves be organizations). For example, auto insurance companies often delegate their customer service commitments to a regional branch, which might further delegate the commitments to a specific agency.

Transfer Responsibility
Intent: A debtor nullifies its original commitment by delegating it to another party, and is no longer concerned with the delegated commitment's satisfaction or violation

Motivation: If the customer delegates to his credit card bank to pay the merchant, the subsequent interactions for the payment are between the bank and the merchant; the customer need no longer be involved
Applicability: When the delegatee and creditor have a business relationship
Consequences: The creditor must accept the delegation, and perhaps seek proof that the delegatee accepts it; the delegation may be risky for the creditor
Implementation: Figure 3(a). Upon *delegate*, original is null whereas delegated becomes active
Known uses: When an airline "endorses" a ticket over to another airline based on a passenger's request, the second airline becomes responsible for transporting the ticketed passenger

Retain Responsibility (in Delegation)

Intent: A debtor delegates its commitment but doesn't give up responsibility for its ultimate satisfaction
Motivation: The merchant delegates its commitment to ship goods to a shipping service, but remains committed to the customer to have the goods delivered; discharging the delegated commitment discharges the original pending commitment
Applicability: When the delegatee and creditor don't have a longstanding business relationship
Consequences: The creditor is safe because the delegator remains responsible; this pattern enables and coheres with Escalate and Withdraw patterns
Implementation: Figure 3(b). Upon *delegate*, original becomes pending whereas delegated becomes active
Known uses: When an insurance company delegates a claimant's auto repair work to a mechanic, it remains responsible if the mechanic fails to make adequate repairs

Escalate (Delegated Commitment)

Intent: The delegatee bailing out of a delegated commitment reactivates the original commitment
Motivation: If a shipper fails to deliver the goods, the merchant becomes responsible again
Applicability: When the delegatee may not provide guaranteed service
Consequences: The creditor would be the instigator
Implementation: Figure 3(c). If delegated is escalated and goes to null, original goes from pending to active, thus reactivating it
Known uses: When a customer pays with a check, it delegates its commitment to pay to the bank; if the bank fails to pay (say because of insufficient funds), the escalation reactivates the customer's original commitment to pay

Withdraw Delegation

Intent: Withdrawing a delegation reactivates the original commitment
Motivation: When a delivery is unacceptably delayed, the merchant may withdraw a delegation from the first shipper and delegate it to another shipper
Applicability: When costs can be saved by cutting off an unproductive delegation
Consequences: Should allow withdrawal to occur only if delegated has not been satisfied: this would arise automatically when this pattern is applied in conjunction with Retain Responsibility, as a result of which satisfaction of delegated would bring original out of its pending state
Implementation: Figure 3(d). Upon *withdraw*, original once again becomes active whereas delegated becomes null
Known uses: When an airline with an over-booked flight delegates its commitment (to transport a passenger) to another airline. If the second airline's flight is excessively delayed due to weather, the first airline may reactivate its commitment to transport the passenger

Divide Labor

Intent: Split service commitment among service providers
Motivation: Assign subtasks to subcontractors
Applicability: Whenever a service can be partitioned based on capabilities or capacity
Consequences:

Implementation: Figure 3(e) with retained responsibility. When the delegatees *discharge* their commitments, original becomes satisfied. Similarly for transferred responsibility
Known uses: Common, RosettaNet distribute work (PIP7B1)

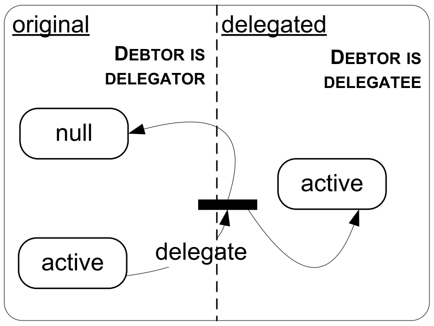
Contextual Patterns

The business context of a service engagement dictates the rules of encounter to which it is subject. For example, eBay users are subject to the terms and conditions of the eBay marketplace, such as that users may not attempt to place false bids. More pertinently, the rules for dispute resolution are also contextual in nature. Each of these patterns involves the three participants: debtor, creditor, and context, with the context explicitly acting as a debtor of a *metacommitment* (whose precondition and condition involve commitments). The context has the power to create and manipulate commitments among the agents in its scope. Metacommitments provide guarantees to the participants.

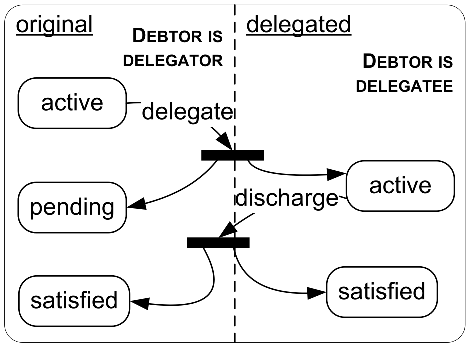
Whereas all commitments involve a context, in the contextual patterns, the context agent itself features as a debtor or creditor. Often, in contextual patterns, the context commits to another party that if some conditions prevail, it will cause a specified commitment to transition to a suitable state.

Penalize
Intent: To impose a penalty upon a party that violates its commitment *Motivation:* The context imposes another commitment upon a debtor in violation. For example, if the original commitment means paying \$10 by Monday, the penalty commitment could mean paying \$11 by Tuesday. Or, if the original means delivering the goods, the penalty could mean refunding the deposit and an additional 10% *Applicability:* When the engagement is subject to a regulatory agency *Consequences:* The context has the means to determine that the requisite conditions hold; it has power over the debtor such as removing it from a marketplace or voiding its license to operate *Implementation:* Figure 4(a). If original is violated, then the precondition of the context's commitment context is met, which in turn makes penalty active *Known uses:* Uniform Commercial Code

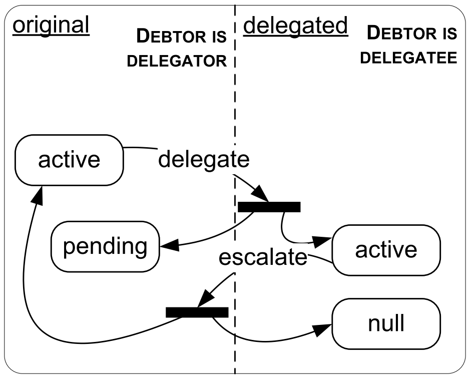
Revert offer
Intent: To enable a party to back out of a transaction *Motivation:* A customer commits to paying for some goods, which are delivered; if the customer returns the goods before paying, he is released from paying; if the customer has paid, the merchant should refund the payment *Applicability:* As in Penalize *Consequences:* As in Penalize *Implementation:* Figure 4(b). An *undo(precondition)* undoes the precondition of the offer. If progress is in state base, then the debtor is released (progress becomes null)—no further action is needed. If progress is satisfied, then *undo(precondition)* creates revert *Known uses:* Uniform Commercial Code



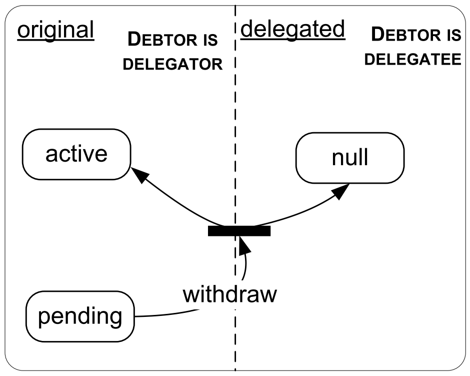
(a) Transfer responsibility



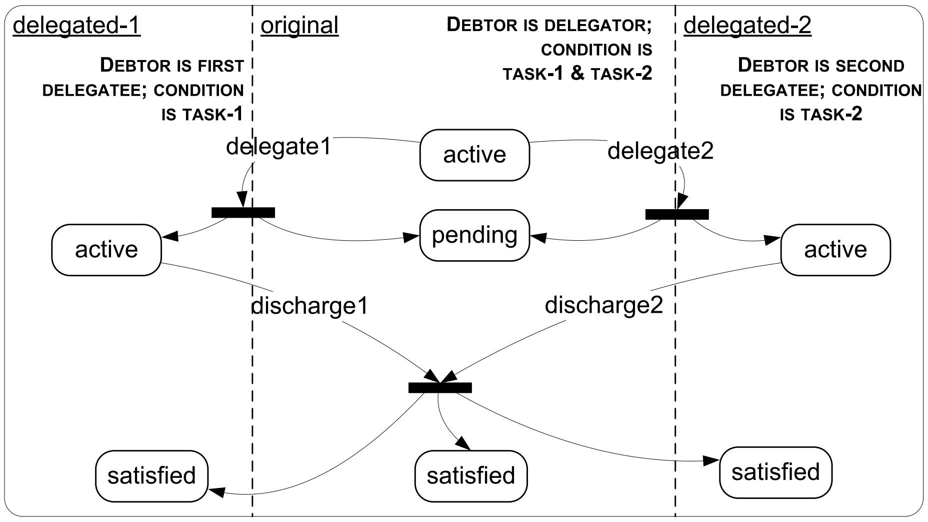
(b) Retain responsibility



(c) Escalate



(d) Withdraw delegation

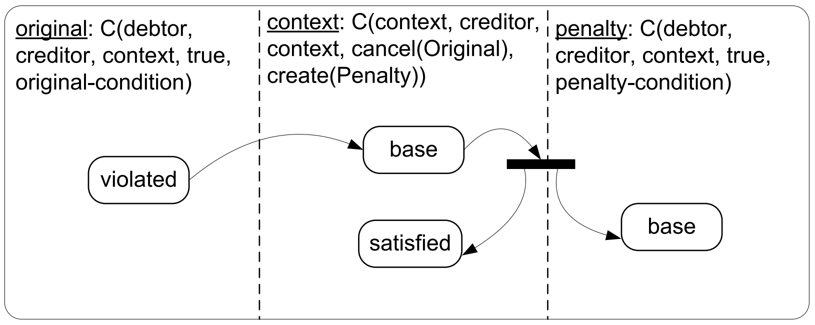

delegated-1
DEBTOR IS FIRST DELEGATEE; CONDITION IS TASK-1
original
DEBTOR IS DELEGATOR; CONDITION IS TASK-1 & TASK-2
delegated-2
DEBTOR IS SECOND DELEGATEE; CONDITION IS TASK-2
delegate1
active
delegate2
active
pending
active
discharge1
discharge2
satisfied
satisfied
satisfied

(e) Divide labor

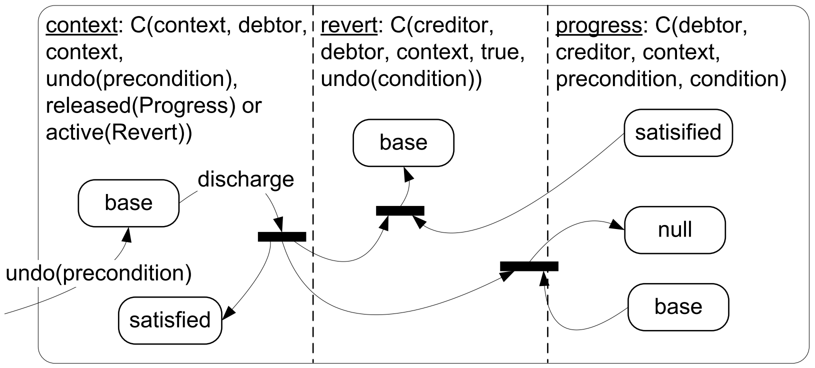
Figure 3: Structural patterns

Applying the Patterns

To design a service engagement using the above patterns requires three steps. One, identify the commitments regarding the services involved. Two, apply selected patterns to appropriate commitments.


original: C(debtor, creditor, context, true, original-condition)
context: C(context, creditor, context, cancel(Original), create(Penalty))
penalty: C(debtor, creditor, context, true, penalty-condition)
violated
base
satisfied
base

(a) Penalize


context: C(context, debtor, context, undo(precondition), released(Progress) or active(Revert))
revert: C(creditor, debtor, context, true, undo(condition))
progress: C(debtor, creditor, context, precondition, condition)
base
discharge
undo(precondition)
satisfied
base
satisified
null
base

(b) Revert offer

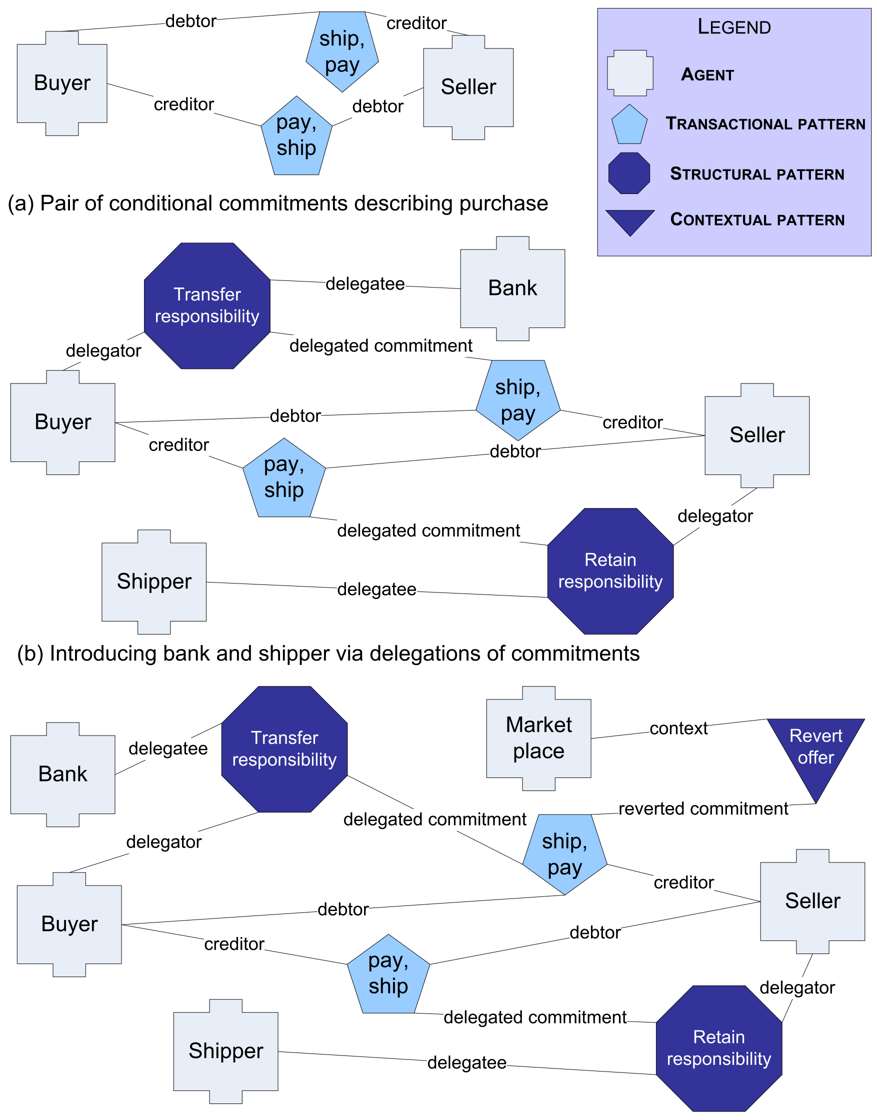
Figure 4: Contextual patterns

Three, map the operations occurring in the patterns to the business actions of the engagement.

Figure 5 illustrates how the above patterns apply in modeling our purchase example. Part (a) shows the partner roles and their commitments toward one another: the customer offers to pay if the merchant ships him the goods; the merchant offers to ship the goods if the customer pays. Part (b) introduces a bank and shipper; the customer delegates the payment commitment to the bank and the merchant delegates the shipping commitment to the shipper; the two apply different structural patterns. Part (c) applies a contextual pattern enabling refunds upon return. Notice that additional business requirements are accommodated simply by applying additional patterns; the existing patterns stay as is. The CSOA patterns may need to be supplemented with operational constraints such as data flow and event ordering among business actions, for example, to require that payments should precede shipping.

By contrast, traditional approaches such as BPMN are based solely on such operational constraints. The resulting flows could be quite complex: imagine trying to achieve the effects of the patterns in Part (c) purely through control and data flows. Not only do they hide the business meaning, they complicate accommodating additional business requirements: even a simple change can lead to many additional intricate changes in the existing control and data flows.

To instantiate an engagement, business partners adopt the specified roles and perform the services and other business actions specified. The patterns refer to several explicit actions, including *create*, *delegate*, *assign*, *release*, and *update*. Each such action is governed by the corresponding partner's policy; at enactment such policies would determine what computations occur. Our proto-



(a) Pair of conditional commitments describing purchase

(b) Introducing bank and shipper via delegations of commitments

(c) Allowing buyer to skip payment or get a refund upon returning goods

Figure 5: Successive CSOA models of a purchase service engagement

type tools map commitment patterns to computations [3] and produce role skeletons, which can be used to implement agents who can participate in an engagement [4].

Traditional models emphasize operationalizations by describing engagements in terms of computations (tasks or messages). They are unwieldy because even simple business requirements can lead to many possible operationalizations. They lack a formal representation of business meaning which is relegated to documentation. We need the operationalizations, of course, but the modeler should be concerned with business meanings, not with low-level operationalizations.

The above patterns describe abstract possibilities. However, applying patterns involves matching them to the concrete business realities of a service engagement. For example, a transactional pattern allowing cancellation would make sense only if a commitment can be reasonably canceled.

And, it may not be possible to delegate a commitment if the intended delegatee would not accept the delegation. Lastly, the context may not be able to ensure that an agent will discharge any commitments created by the context. In general, the above patterns would work best when there is a suitable prior business or legal relationship among the parties involved. The patterns can guide the creation of the appropriate relationships or constraints so that desired service engagements are realized.

SOAs as Architectural Styles

An architectural style specifies a family of configurations of *components* and *connectors* subject to stated *constraints* [8].

In these terms, existing SOAs are an architectural style whose major components are (service) provider and consumer, and whose connector is a protocol by which a consumer invokes a provided service. (For simplicity, we ignore registries, and service publication and discovery.) A practical SOA includes specialized components and connectors, such as for resource management and other enterprise functions (identity, billing, and so on), and imposes additional constraints so appropriate components interoperate with the management and enterprise components.

Benatallah *et al.* propose patterns called *business-level interfaces and protocols* [9]. However, their patterns ignore business meanings (like CDL and BPEL), thereby leading to rigid interoperation. For example, if a message interface specifies that a customer should make a payment subsequent to the receipt of goods, then a service realizing such an interface must behave accordingly. It ought not to take any liberties such as reversing the order of the messages, interposing other messages, or introducing another party such as a payment agency. But, real-life service engagements typically presume such flexibility. Limiting flexibility subverts the services vision because it creates avoidable friction in the web of value.

Table 1: A comparison of commitment-based and existing SOAs with respect to the main elements of an architectural style (following [8])

Elements	**Existing SOAs**	**Commitment-Based SOA**
Components	Service provider and consumer	Business service provider and consumer agents
Connectors	Operations and message patterns (in, out, in-out, out-in)	Commitment patterns
Invariants	Match operation and message signatures	Each party knows only the identity of parties with whom it features in a commitment
Model	Control and data flow	Operations on commitments

The motivation for considering business meaning is to improve the flexibility of enactments and the naturalness, maintainability, and reusability of service specifications. Table 1 contrasts commitment-based and existing SOAs. Thus, CSOA is not a unique style but has many flavors depending on the patterns selected. Such flexibility is necessary to support the nuances of service engagements. The primary constraint on a sound implementation of CSOA is that at runtime all commitments are eventually null or satisfied.

The reader may reasonably wonder: if CSOA is so different why is it still a SOA? The answer is twofold. One, CSOA is centered on services and is, arguably, more true to the services vision than existing SOAs. Two, CSOA doesn't seek to replace existing SOAs and their implementations. CSOA is conceptualized in terms of business service engagements, but would be realized through conventional means. For example, the service engagements modeled in CSOA would translate into business processes expressed in BPMN.

Model-Driven Architecture[9] (MDA) provides a useful way to think of the relationship between CSOA and existing SOAs. In MDA terms, CSOA is a Computation Independent Model whereas existing SOAs are Platform Independent Models. In other words, the move to CSOA would represent the step—often repeated in computer science—of moving from lower to higher abstractions. Because commitments are computation independent, yet lend themselves to rigorous operationalization, CSOA can help bridge the well-recognized gap between business and IT [10]. Others have begun to recognize the importance of high-level abstractions, but their work still employs operational abstractions.[10]

Kumaran [11] presents four abstraction layers for enterprise modeling: strategy (business considerations), operation (business functions conceptualized via tasks and artifacts), execution (analogous to existing SOAs), and implementation. CSOA would help extend Kumaran's operation layer to multienterprise service engagements, and commitment patterns would provide richer representations that facilitate modeling enterprise operations perspicuously and reusably.

Because of the subtleties of real-life service engagements, no finite set of patterns would be provably complete. This is analogous to object-oriented design patterns, which are numerous and unbounded even though the underlying programming languages need only a few primitives.

However, despite their subtlety, service engagements for the most part exhibit regularities in how their transactions, structures, and contexts are applied. Consequently, a reasonably small set of patterns can help describe a large number of practical engagements. Thus our main contributions are introducing a SOA that gives primacy to business interactions and showing how to formalize the concomitant patterns that provide an expressive vocabulary for modeling service engagements.

Typical service engagement models would include several of the above patterns applied in routine ways. Thus *aggregate service patterns*, which capture best practices in designing service engagement, may potentially be abstracted, and applied in designing new engagements using CSOA.

References

[1] Munindar P. Singh. Distributed enactment of multiagent workflows: Temporal logic for service composition. *Proc. 2nd Intl. Joint Conf. Autonomous Agents & MultiAgent Systems*, pages 907–914, 2003.

[2] Wil M. P. van der Aalst and Maja Pesic. DecSerFlow: Towards a truly declarative service flow language. *Proc. 3rd Intl. Workshop Web Services and Formal Methods*, *LNCS* 4184, pages 1–23. Springer, 2006.

[3] Amit K. Chopra and Munindar P. Singh. Contextualizing commitment protocols. In *Proc. 5th Intl. Joint Conf. Autonomous Agents & Multiagent Systems*, pages 1345–1352, 2006.

[9] http://www.omg.org/mda

[10] http://www.ip-super.org

[4] Nirmit Desai, Ashok U. Mallya, Amit K. Chopra, and Munindar P. Singh. Interaction protocols as design abstractions for business processes. *IEEE Trans. Software Engineering*, 31(12):1015–1027, December 2005.

[5] Ahmed K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1992.

[6] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[8] Mary Shaw and David Garlan. *Software Architecture*. Prentice-Hall, 1996.

[9] Boualem Benatallah, Fabio Casati, Farouk Toumani, Julien Ponge, and Hamid R. Motahari Nezhad. Service Mosaic: A model-driven framework for web services life-cycle management. *IEEE Internet Computing*, 10(4):55–63, 2006.

[10] Howard Smith and Peter Fingar. *Business Process Management*. Megan-Kiffer Press, 2002.

[11] Santhosh Kumaran. Model-driven enterprise. *Proc. Global Enterprise Application Integration Summit*, pages 166–180, 2004.

Authors

Munindar P. Singh is a Professor of Computer Science at NCSU. He is a former EIC of *IEEE Internet Computing*. Singh coauthored the textbook *Service-Oriented Computing: Semantics, Processes, Agents*.

Amit K. Chopra is a doctoral candidate in Computer Science at NCSU. Chopra's interests include service-oriented architectures and multiagent systems.

Nirmit Desai is a postdoctoral fellow in Computer Science at NCSU. Desai's interests include cross-organizational business processes.