

Modeling Flexible Business Processes^{*}

Amit K. Chopra, Ashok U. Mallya, Nirmal V. Desai, and Munindar P. Singh

Department of Computer Science, North Carolina State University

Abstract. Current approaches of designing business processes rely on traditional workflow technologies and thus take a logically centralized view of processes. Processes designed in that manner assume the participants will act as invoked, thus limiting their flexibility or autonomy. Flexibility is in conflict with both reusability and compliance.

We propose a methodology to build processes from declarative commitment-based protocol specifications and to enact them in a distributed manner. Because protocols are publishable, reusable specifications of interaction and commitments can be reasoned about, this approach enables software reuse, improved autonomy through flexibility, and more robust compliance verification.

We present an operational semantics of protocols and commitments in the π -calculus that better supports contextualized reasoning. Reasoning about commitments leaves protocols reusable and improved process flexibility.

1 Introduction

The modeling and enactment of business processes has received considerable attention in the research community. Cross-enterprise business processes involve a number of components that are independently designed and configured and represent the interests of autonomous parties and yet have to interact coherently. One challenge that arises is to reconcile interoperation with the autonomy of the partners. Another challenge is to make business processes easy to put together from reusable components. Efforts in this area include OWL-S [4], BPEL [1], and XPDL [19]. While these efforts allow the specification and enactment of a business process, they specify the implementation of a process rather than the interactions that are expected of it. More specifically, they rely on flow abstractions that support the perspective of only one participant. They, therefore, support neither reusability or flexible execution in the face of exceptions or opportunities that are a reality in dynamic and open systems.

Interactions in business processes are typically long-lived so that they can be organized in the form of protocols. Protocols offer the level of abstraction that naturally supports local perspectives as they specify the interaction (the *what*) rather than implementation (the *how*). Thus, business protocols naturally maximize the autonomy of the participants. This paper presents an approach for developing business processes that (1) can be built from reusable protocols; (2) is agent-based, implying decentralization; and, (3) affords the agents flexibility in handling exceptions and exploiting opportunities.

^{*} The first three authors are students. This research was supported by the NSF under grant DST-0139037 and by DARPA.

1.1 Challenges in Process Design

We recognize commitments as important in giving a semantics to protocols. As agents interact, they create and manipulate commitments. Previous work has developed a declarative commitment-based semantics for protocols [2, 20] where representing and reasoning about commitments leads to enhanced flexibility in protocols. This paper delves deeper into the design of business processes. Business process design offers challenges that commitment-based protocols can address naturally. First, reusability is in tension with flexibility. Second, compliance is in tension with flexibility. We motivate each in the following.

Reusability Vs. Flexibility For a protocol to be reusable, it should be well-encapsulated and its semantics should be unambiguous. In other words, a protocol should be specified as independently as possible of the context in which it will be used. This context could be the society in which the agent exists in or the physical location of the agent or the other processes in which an agent participates. Adding such context-dependent computations to a protocol would make it unwieldy and non-reusable. However, exception handling is inherently context-sensitive.

Thus reusability is in tension with the ability of an agent to handle exceptions and exploit opportunities that might arise during the enactment of the protocol. (Of course, exceptions that occur routinely and depend on the nature of the protocol could be added to the protocol, but that means they are treated like normal behavior.)

The approach proposed in this paper leaves the protocol reusable, but at the same time allows the agent maximum autonomy in handling exceptions and exploiting opportunities. Giving protocols a commitment-based semantics plays a central role in this scheme.

Compliance Vs. Flexibility Business process agents have two main components, protocols and business policies (see Figure 2). The protocol prescribes the interaction that should take place irrespective of an agent's policies, whereas policies control the interaction, presumably in a way that is compatible with the agent's interests. Policies can be entirely internal or be dependent on the agent's context. The policies generally differ from agent to agent, whereas protocols tend to be reused. Policies and protocols are specified independently of each other. An agent may comply with a protocol completely, but will only have simplistic policies that are unable to handle dynamism. Similarly, an agent could completely respect its policies and therefore, be flexible, but might end up *utterly* violating the protocol. This paper shows how we can strike a balance between compliance and flexibility by reasoning about commitments.

1.2 From Declarative to Operational Semantics

Our design methodology involves taking declarative commitment-based specifications of protocols and extracting operational specifications of role skeletons from them (see Figure 1). Agents are built out of business policy specifications and role skeletons. Policies are used to control the interaction. While a declarative semantics is appropriate for

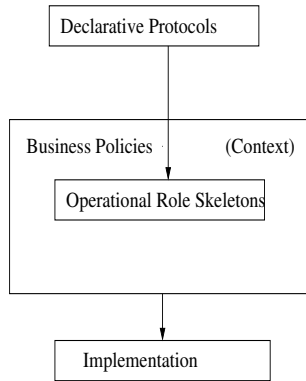


Fig. 1. Abstraction Levels

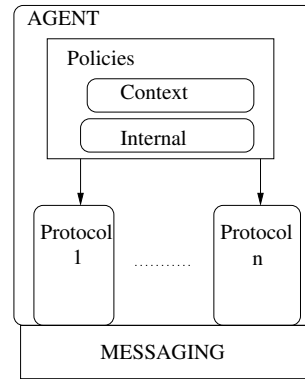


Fig. 2. Agent Construction

protocol specification, an operational semantics is more appropriate for role specification. This is because role skeletons are local in nature and the operational semantics better captures the interaction between roles. More importantly, declarative specifications allow for partial descriptions of systems which is good for global specifications like protocols; the rest of the system description is fleshed out with respect to a context, that is, in an operational setting. The context is accommodated by specifying the business policies of the agent. The operational specifications can then be used to reason about an agent’s compliance, both with its policies and its protocol roles. This is significant because a fully compliant agent would exhibit less autonomy than a non-compliant one.

1.3 Contributions

This paper presents an operational model of commitments and business processes based on the π -calculus. The π -calculus is a process algebra for modeling concurrent processes whose *configuration*, that is, communication links, may change at run time. To avoid ambiguity with business processes, we write π -*process* to refer to a π -calculus process. The most notable contributions of our approach are listed below.

- Business processes can be built from reusable protocols.
- Reasoning about commitments alleviates the tension between reusability and compliance, and flexibility.
- This leads to the development of robust business process agents.

Our proposed approach models a commitment itself as a π -process. The commitment maintains communication channels with the agents that participate in it. Our operational characterization of commitments leads to flexible modeling of the commitment life-cycle. For example, the partial discharge of a commitment can be modeled. More importantly, our modeling alludes to the need for even more sophisticated modeling of commitments to add to the flexibility of protocols.

1.4 Organization

The rest of the paper is organized as follows. Section 2 introduces the basic notions of commitments. Section 3 describes our conceptual model of building business processes. Section 4 sketches the π -calculus. Section 5 formalizes the commitments and their operations in the π -calculus. Section 6 models the NetBill protocol [3] in the π -calculus. Section 7 talks about reasoning with commitments. Section 8 discusses relevant literature and future work.

2 Commitments

Commitments have been identified as a key abstraction in the modeling of agent interaction protocols and languages [15, 8, 6, 2, 7]. As agents interact, they create and manipulate commitments. In simple terms, a commitment represents a directed obligation from one agent to another for maintaining or achieving a condition. Knowing what commitments exist helps an agent plan its actions and leads to coordination between agents. Another advantage of commitments is that they give a social semantics to interaction.

A number of operations may be performed on commitments. Following [16], we formally define commitments and the operations that can be performed on them.

Definition 1. A base-level commitment $C(x,y,G,p)$ binds a debtor x to a creditor y for fulfilling the condition p in context G .

Definition 2. A conditional commitment $CC(x,y,G,p,q)$ denotes that if a condition p is brought about, then the commitment $C(x,y,G,q)$ will hold.

Both commitments and conditional commitments are created in a context G , which can be thought of as an institution or society whose rules are binding on the agents that participate in it. The context also defines the meanings of the terms used. Since we will only informally talk about the context, we omit G to reduce clutter. Below we list the commitment operations.

- *Create*(x,y,p) creates a new commitment $C(x,y,p)$.
- *Discharge*(x,y,p) discharges the existing commitment $C(x,y,p)$ so that it no longer holds. A discharge is done only when the condition p starts to hold, i.e., the commitment is satisfied.
- *Cancel*(x,y,p) cancels the existing commitment $C(x,y,p)$ so that it no longer holds. Only debtors can cancel a commitment.
- *Delegate*(x,y,p,z) delegates the commitment $C(x,y,p)$ to a new debtor z . More specifically, the original commitment $C(x,y,p)$ no longer holds and a new commitment $C(z,y,p)$ is created in its place.
- *Assign*(x,y,p,z) assigns the commitment $C(x,y,p)$ to a new creditor z . More specifically, the original commitment $C(x,y,p)$ no longer holds and a new commitment $C(x,z,p)$ is created in its place. Only a creditor can do an assign.
- *Release*(x,y,p) releases the debtor x from the commitment $C(x,y,p)$ so that the commitment no longer holds. Only a creditor can do a release.

3 Protocols for Business Processes

We look at protocols as reusable specifications of business interactions. Before we present a conceptual model for building processes from protocols, we explain in detail our motivation for using protocols as building blocks for processes.

- Business processes tend to be complex and implementation-specific. They are therefore, not amenable for reuse. On the other hand, protocols are declarative publishable specifications describing only the interaction and can therefore, be reused. Protocols themselves can be complex, but a process that uses a complex protocol will be even more complex.
- Representing commitments leads to flexible protocols. Flexible protocols naturally maximize the autonomy of interacting parties. Using such protocols to design processes will lead to autonomy-preserving business processes while ensuring interoperation at the same time.
- Since protocols have a commitment-based semantics, this paves the way to formally reason about protocols. From the design point of view, it allows creation of newer protocols by specializing and aggregating existing ones. For example, payment by credit card is a specialization of a general payment protocol. Similarly, ordering, shipping and payment protocols can be spliced together to form a complete trading protocol.
- Building business processes around protocols will lead to modular business process where interaction and policy are cleanly separated.

Figure 3 presents a conceptual model of how to build processes from protocols. A *business protocol* is a declarative specification that specifies the business interactions. The protocol skeletons *P-Skels*, one for each role in the protocol, are extracted from the specification. An agent is an implementational entity representing a business partner that adopts one or more roles in one or more protocols. The *C-Skel* corresponds to composition of the *P-Skels* of the adopted roles. This composition may be policy based. The *C-Skel* represents the local flow enacted by the agent. A business process aggregates the local flows of the agents participating in it.

4 The π -Calculus

The π -calculus [11–13] is a process algebra for modeling concurrent processes whose configurations, that is, communication links may change as the processes execute. In the π -calculus, the fundamental unit of computation is the transfer of a communication link between two processes. Intuitively, the communication link is like an access to a resource. The simplicity of the π -calculus arises from the fact that it includes only two kinds of entities: names and agents (processes). These are sufficient to rigorously define interactional behavior. Interaction corresponds to a handshake between two processes and involves the output of a link by one process and simultaneous receipt of the link as input by another process. For this paper, we shall limit ourselves to the basic or synchronous π -calculus. The following briefly presents the π -calculus language and some examples.

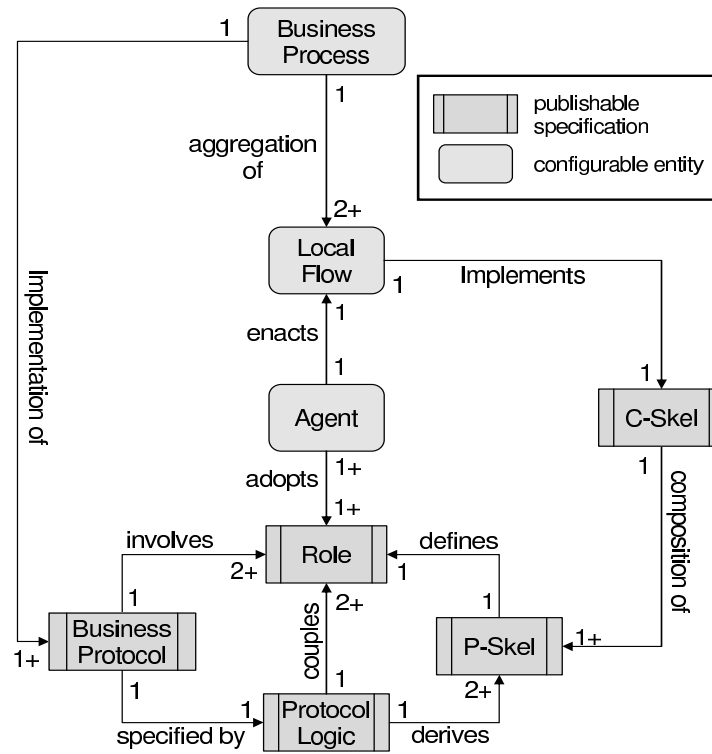


Fig. 3. Conceptual Model

4.1 Process Syntax

The language of the π -calculus consists of prefixes and process expressions, as summarized in Table 1. Below we explain the language constructs in the same order as Table 1.

Prefixes are of the following kinds:

- The output prefix $\bar{a}x$ means that x is sent along the channel a .
- The input prefix $a(x)$ means that the channel a can be used to receive input and binds this input to x .
- The silent τ means that nothing observable happens.

The process expressions are as follows:

- 0 represents the nil-process.
- $\alpha.P$ does the action represented by prefix α and changes to P .
- $P + Q$ represents the sum-nondeterminism, that is, do either process P or process Q .
- $P|Q$ represents that process P and process Q execute in parallel.

Prefixes	$\alpha ::=$	$\bar{a}x$	(Output)
		$a(x)$	(Input)
		τ	(Silent)
Agents	$P ::=$	0	(Nil)
		$\alpha.P$	(Prefix)
		$P + P$	(Sum)
		$P P$	(Parallel)
		$[x = y]P$	(Match)
		$[x \neq y]P$	(Mismatch)
		$(new\ x)P$	(Restriction)
		$!P$	(Replication)
		$A(y_1, \dots, y_n)$	(Identifier)
Definitions	$A(x_1, \dots, x_n) \stackrel{def}{=} P,$	$(where\ i \neq j \Rightarrow x_i \neq x_j)$	

Table 1. π -calculus Syntax

- $[x = y]P$ or *match* represents the process that changes to P if $x = y$. Mismatch is the opposite, i.e., it checks $x \neq y$.
- $(new\ x)P$ means that the variable x is declared as a new name local to process P and bound in P . It is not visible outside of P .
- $!P$ represents an unbounded number of copies of the process P . Formally, $!P \stackrel{def}{=} P|!P$.
- $A(y_1, \dots, y_n)$ represents the instantiation of a defined agent.
- $A(x_1, \dots, x_n) \stackrel{def}{=} P$ ($where\ i \neq j \Rightarrow x_i \neq x_j$) represents the declaration of a process A in terms of process P . One can think of it as a method declaration in traditional procedural programming.

The input prefix and the *new* operator bind the names. For example, in a process $a(x).P$, the name x is bound, but a is not. This is similar to the λ -calculus. In $(new\ x)P$, x is considered to be bound. As in the λ -calculus, α -conversion might be necessary to avoid capture of free names. The free names of a process indicate the linkage to the environment.

5 Commitments in the π -Calculus

In the π -calculus, processes are also known as agents. However, from now on, we use the term *agent* for a process capable of becoming a debtor or creditor for a commitment. The commitment operations have constant names DISCHARGE, CANCEL, and so on. The environment of a process consists of all other processes in the system.

5.1 Channels

We propose that a commitment be represented in the π -calculus via a process that has four channels, which we list below (See Figure 4):

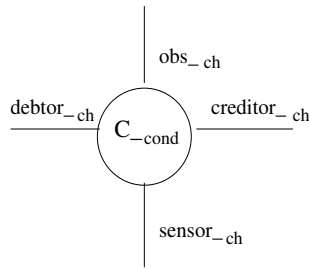


Fig. 4. Commitment channels

- $debtor_{ch}$ represents a link to the commitment’s debtor. The debtor sends the name of the commitment operation it wishes to perform, one of *Cancel*, *Delegate* or *Discharge* on this channel. The success of the operation depends on how the *Guard* of the commitment evaluates. We explain the *Guard* shortly.
- $creditor_{ch}$ represents a link to the commitment’s creditor. The creditor sends the name of the commitment operation it wishes to perform, one of *Assign* or *Release*, on this channel. Once again, the success of the operation depends on how the *Guard* of the commitment evaluates.
- $sensor_{ch}$ represents a link to the commitment’s environment on which the commitment receives the condition (*payment*, for instance) it is interested in. The condition could correspond to the condition of the commitment or a prerequisite for some commitment operation. The *Guard* evaluates the received condition; intuitively, it is guarding the commitment operations.
- obs_{ch} Some processes other than the creditor and debtor might be interested in knowing what operation took place in a particular commitment. Each such process uses the obs_{ch} channel to subscribe for the operation that it is interested in. Then, obs_{ch} returns to the subscribing process a private channel, on which notifications are delivered.

An alternative model is possible where, along with the operation name an agent wishes to perform, it also sends the event on the $debtor_{ch}$ (or $creditor_{ch}$ depending on what role the agent plays in the commitment). Such a model would mean that only those conditions which are received from the debtor (or creditor) can be evaluated. In our model, the use of $sensor_{ch}$ separate from creditor and debtor channels allows the condition to be received from any process. For example, this flexibility is useful for modeling third-party verification of satisfaction of conditions. The case where the sensor receives input from the debtor (or creditor) is a special case of this model.

The intuition behind the obs_{ch} is similar to the intuition behind $sensor_{ch}$. The obs_{ch} gives the model independence from having to receive notifications solely from the agents. Both, the $sensor_{ch}$ and obs_{ch} , facilitate modular designs since the agents are no longer hard-coded.

Normal Commitment Process

$$C(\text{debtor}_{ch}, \text{creditor}_{ch}, \text{cond}, \text{sensor}_{ch}, \text{obs}_{ch}) \stackrel{def}{=} (\text{new } y)((\text{DebtorOp} + \text{CreditorOp}) \mid \text{Subscribe})$$

Special Commitment Process

$$C(\text{debtor}_{ch}, \text{creditor}_{ch}, \text{cond}, \text{sensor}_{ch}, \text{obs}_{ch}, y) \stackrel{def}{=} (\text{DebtorOp} + \text{CreditorOp}) \mid \text{Subscribe}$$

Table 2. Commitment process definition

$$\begin{aligned} \text{DebtorOp} &\equiv \text{debtor}_{ch}(op).([\text{op} = \text{DISCHARGE}] \text{Discharge} + [\text{op} = \text{CANCEL}] \text{Cancel} \\ &\quad + [\text{op} = \text{DELEGATE}] \text{Delegate}) \\ \text{CreditorOp} &\equiv \text{creditor}_{ch}(op).([\text{op} = \text{RELEASE}] \text{Release} + [\text{op} = \text{ASSIGN}] \text{Assign}) \\ \text{Discharge} &\equiv \text{Guard}.\overline{\text{creditor}_{ch}}\text{DISCHARGE}.\!(\overline{y}\text{DISCHARGE}) \\ &\quad + \text{NotGuard}.\overline{C}(\text{creditor}_{ch}, \text{debtor}_{ch}, \text{cond}, \text{sensor}_{ch}, \text{obs}_{ch}, y) \\ \text{Cancel} &\equiv \overline{\text{creditor}_{ch}}\text{CANCEL}.\!(\overline{y}\text{CANCEL}) \\ \text{Delegate} &\equiv \text{debtor}_{ch}(\overline{\text{delegatee}_{ch}}).\overline{\text{delegatee}_{ch}}.\text{debtor}_{ch}.\overline{C}(\text{debtor}_{ch}, \text{creditor}_{ch}, \text{cond}, \\ &\quad \text{sensor}_{ch}, \text{obs}_{ch}, y).\overline{\text{creditor}_{ch}}\text{DELEGATE}.\!(\overline{y}\text{DELEGATE}) \\ \text{Subscribe} &\equiv (\text{new } x)\text{obs}_{ch}(\text{operation})\text{obs}_{ch}x.\!(y(op).[\text{operation} = \text{op}]\overline{x}) \\ \text{Guard} &\equiv \text{sensor}_{ch}(\text{value})[\text{value} = \text{cond}] \\ \text{NotGuard} &\equiv \text{sensor}_{ch}(\text{value})[\text{value} \neq \text{cond}] \end{aligned}$$

Table 3. Constituent processes

5.2 Commitment Process

Now that we have introduced the channels that are used and the intuitions behind their existence and usage, we can present the model in detail. We describe this model in the context of how various operations on commitments—whose formal expressions are given in Table 2 and 3—would be realized through it.

Create. All this while, we have not talked explicitly about modeling the creation of a commitment. The reason is that we model commitment as a parametric process. Invoking this process corresponds to creating a commitment. For reasons explained below, we consider two variations on the parametric process for creating commitments. Table 2 lists the formal descriptions of the two processes.

Consider the normal commitment process: $C(\text{debtor}_{ch}, \text{creditor}_{ch}, \text{cond}, \text{sensor}_{ch}, \text{obs}_{ch})$. Ignoring *Subscribe* for the time being, the definition says that either a creditor operation or a debtor operation can be performed depending on the channel on which the input arrives, i.e., creditor_{ch} or debtor_{ch} . *cond* represents the condition of the commitment. *CreditorOp* and *DebtorOp* represent the creditor and debtor operations, respectively. The creditor_{ch} channel is read for name of the operation requested and the operation is attempted. Similarly, debtor_{ch} is read for debtor operations.

Discharge. Here the *Guard* simply checks if the value read on sensor_{ch} matches the condition. First the *Guard* is evaluated. If a match is successful, then *Discharge* is

successful and the creditor is notified. If the match fails, i.e., *NotGuard* succeeds, then the commitment is recreated.

Cancel. The creditor is notified of the cancel operation.

Delegate. The delegatee is passed $debtor_{ch}$, the commitment is recreated, and the creditor is notified of the operation.

Assign and Release. For simplicity and to save space, we do not explicitly model *Assign* and *Release* as these are conceptually similar to *Delegate* and *Cancel*, respectively.

To function cleanly, the commitment operations require some auxiliary processes. *Guard* and *NotGuard* were described above. A more important process is *Subscribe*. *Subscribe* enables other processes to receive notifications from the commitment (by subscribing to such notifications). A subscription is created by sending on the obs_{ch} the name of the operation that the subscribing process is interested in. A private channel x is returned to the process. When that operation happens on the commitment, it sends a notification to the process on x .

Internal to a commitment, when an operation happens, it sends a notification on y . Note $\overline{y}DISCHARGE$, $\overline{y}CANCEL$ and $\overline{y}DELEGATE$ in Table 3. The notification is modeled as happening an unbounded number of times meaning that enough copies of the notification are always available. An unbounded number is essential because of two reasons:

- The number of processes that may be registered for the same operation is not known in advance and allowing potentially unbounded notifications simplifies our representation. This is realistic in terms of practical implementations.
- In *Subscribe*, the match $[operation = op]$ may fail. In this case, the process must be ready to test another input for forwarding to the subscriber. This is also why in *Subscribe* there is a repetition $!(y(op).[operation = op]\overline{x})$.

In the same vein, there are unboundedly many copies of *Subscribe*. For every registration request on obs_{ch} a fresh copy is, in essence, activated.

The special parametric commitment process shown in Table 2 is used when a commitment operation recreates a commitment. This happens in *Discharge* when the *NotGuard* branch is taken and in *Delegate* when the new commitment reflecting the delegation is created. When recreating a commitment, it is necessary to pass the original y channel since prior subscriptions for notifications must still be honored.

5.3 Modeling Conditional Commitments

$$CC(debtor_{ch}, creditor_{ch}, cond_1, cond_2, sensor_{ch1}, sensor_{ch2}, obs_{ch}) \stackrel{def}{=} (new\ trig_{com}) \\ (\overline{Guard}_1.trig_{com} + \overline{NotGuard}_1.CC(debtor_{ch}, creditor_{ch}, cond_1, cond_2, sensor_{ch1}, \\ sensor_{ch2}, obs_{ch}))|trig_{com}.C(debtor_{ch}, creditor_{ch}, cond_2, sensor_{ch2}, obs_{ch})$$

Table 4. Conditional commitment process

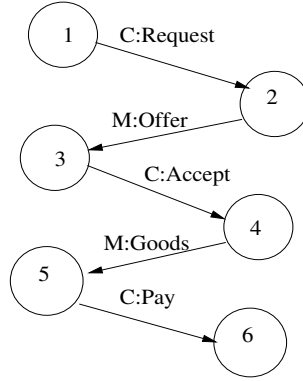


Fig. 5. NetBill

Just like commitments, conditional commitments support all the commitment operations. Likewise, events on the conditional commitment itself may be observed by other interested parties. However, for simplicity, we omit the operations and subtleties. Table 4 shows the corresponding formula. $Guard_1$ checks if the right condition is received on $sensor_{ch1}$. If so, it triggers the creation of the base-level commitment. Otherwise, it recreates the conditional commitment.

6 NetBill in π -calculus

To illustrate our model, we represent the simplified version of the NetBill protocol in the π -calculus. Figure 5 shows the protocol. There are two roles in the protocol; customer and merchant denoted by C and M respectively in Figure 5. The protocol starts with the customer sending a request for offers and ends with the customer sending payment. The commitment is created by the debtor of the commitment and access to the commitment is passed to the creditor. Also, all the input to the sensor channels are coming from agents in this model.

We abbreviate $(new\ x)(new\ y)(new\ z)$ to $new(x\ y\ z)$. Similarly we will abbreviate $x(y).x(z)$ to $x(y\ z)$. We abbreviate output similarly. We also drop the obs_{ch} in the specification as we do not use it. Table 5 shows the encoding of NetBill in π -calculus. cre and deb denote respectively, the creditor and debtor channels of a commitment. cre_{goods} , for example, denotes the creditor channel for goods. sen denotes the sensor channel of a commitment.

We now illustrate the flexibility π -calculus brings to protocol modeling with some examples.

Example 1. Given this specification of NetBill in π -calculus and our encoding of commitments, we now present a deviation from the NetBill protocol that our specification can handle elegantly. When it is time to send the payment, the customer delegates the payment to another agent ccc that represents the credit card company. ccc will even-

$NetBill \equiv$	$(new\ x)(Customer Merchant)$
$Customer \equiv$	$SendRequest.ReceiveOffer.SendAccept.ReceiveGoods.$ $SendPayment.ReceiveReceipt$
$Merchant \equiv$	$ReceiveRequest.SendOffer.ReceiveAccept.SendGoods.$ $ReceivePayment.SendReceipt$
$SendRequest \equiv$	$\bar{x}request$
$ReceiveOffer \equiv$	$x(offer\ cre_{goods}\ sen_{accept})$
$SendAccept \equiv$	$new(accept\ deb_{pay}\ cre_{pay}\ goods\ pay\ sen2_{goods}\ sen_{pay})(\overline{sen_{accept}accept}.$ $\bar{x}(cre_{pay}\ sen2_{goods}) CC(deb_{pay},\ cre_{pay},\ goods,\ pay,\ sen2_{goods},\ sen_{pay}))$
$ReceiveGoods \equiv$	$\overline{cre_{goods}(DISCHARGE)}$
$SendPayment \equiv$	$\overline{deb_{pay}DISCHARGE.sen_{pay}.pay}$
$ReceiveRequest \equiv$	$x(request)$
$SendOffer \equiv$	$new(offer\ deb_{goods}\ cre_{goods}\ accept\ goods\ sen_{accept}\ sen1_{goods})$ $(\bar{x}(offer\ cre_{goods}\ sen_{accept}) CC(deb_{goods},\ cre_{goods},\ accept,\ goods,$ $sen_{accept},\ sen1_{goods}))$
$ReceiveAccept \equiv$	$\overline{x(cre_{pay}\ sen2_{goods})}$
$SendGoods \equiv$	$\overline{deb_{goods}DISCHARGE.sen1_{goods}goods.sen2_{goods}goods}$
$ReceivePayment \equiv$	$\overline{cre_{pay}(DISCHARGE)}$

Table 5. NetBill Process

tually satisfy the commitment and the merchant process could handle the delegation cleanly.

The NetBill process could handle such a deviation because the operations are handled inside the commitment process, not in the agents, and operations basically require nothing more than channel extrusion, i.e, handing off the relevant channels to other π -processes. ■

Example 2. Consider the case where partial discharges are allowed. For example, instead of the customer paying for the goods in one shot, it could send multiple, smaller payments, thereby discharging its commitment to pay in steps. Such a case can be handled by reading the amount of the current payment on the sen_{pay} and implementing *Discharge* differently. The *Guard* in this new *Discharge* would implement arithmetic instead of simple match. If the result of the *Guard* ≥ 0 , then a residual commitment would be created. ■

Figure 6 shows the state of the process in terms of the commitments that exist in NetBill after executing the protocol actions up to *accept*. Note that commitments act as intermediaries between the processes. All the important events or conditions affecting commitments pass through them and their internal logic decides if operations are successful. In other words, they are doing the compliance checking. The implementation of a compliance checker could potentially be derived from the design of the commitment. We leave this to future work.

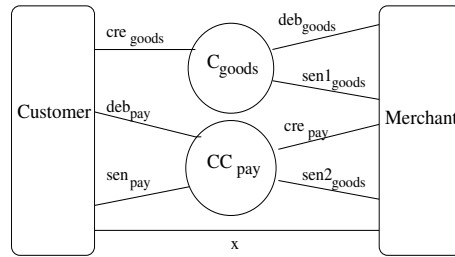


Fig. 6. NetBill Process After Accept

7 Reasoning With Commitments

A protocol is violated if a commitment made in the protocol is violated. For example, if a merchant accepts payment, but fails to ship the goods then it is violating its commitment to send the goods. Policies are a combination of policies inherited from the context and an agent's internal policies. The agent has more leeway in enforcing its internal policies, but it usually must enforce contextual policies. For example, US law prohibits the sale of software with strong encryption to customers outside the US. An agent that sells encryption software will have this law encoded as its contextual policy. Contextual policies will usually encode commitments to an institution. Violation of contextual policies usually result in penalties. The institution in the above example is the US government. An example internal policy is that the agent accepts only credit cards. Note that generally, execution of the commitment operations are also governed by contextual policies. For instance, a *Cancel* would normally be prohibited by the context unless some special circumstances hold.

Now consider the exception that occurs when the merchant is unable to ship the goods after receiving payment from the customer. Because commitments are represented, it knows it is committed to sending the goods and that failure to do so will invite some kind of social penalty. So it decides to either delegate the responsibility of shipping to some other merchant or if that is not possible, it sends a refund. Because delegation is not specified in the protocol, it still represents a weak violation of the the protocol. Refunding the money represents a stronger violation of the protocol. But reasoning about commitments lets it find a somewhat satisfactory solution to this exception. Of course, it is possible that the customer agent might find none of *delegation* or *refund* acceptable. However, the exception handling like this is certainly preferable to the situation where the merchant does not know how to handle a failure to deliver goods and does nothing or aborts the transaction. It is possible to add the 'delegate' and 'refund' computations to the protocol itself, but that would be a manifestation of a particular agent's policy in the protocol, making the protocol unwieldy and less reusable. An example of an opportunity would be when for a large enough transaction, the merchant overrides its policy to only accept credit cards in favor of wire transfers. A contextual policy could be overridden for similar reasons.

To specify this at an architectural level, a π -process encodes the policies of the agent. Before every transition, the protocol skeleton consults with a process called the *commitment-collector* (in the sense of a garbage collector) that looks at the state the role skeleton is in, determines its outstanding commitments in the protocol and takes steps to *execute*, that is, satisfactorily handle those commitments. The commitment-collector could also proactively poll the state of the skeleton depending upon the timeouts of commitments. We can thus view the agent as a virtual commitment machine that executes commitments.

The most important point to note here is that we are separating interaction and control. Control resides with the commitment-collector whereas the role skeleton just carries out the interaction. This allows the reusability of the protocol in another context and keeps the handling of exceptions and opportunities in the commitment-collector.

8 Discussion

This paper presents a vision of business process design. Current business processes end up being rigid and limit the participating agents' autonomy. Although current techniques give a semantics to the data and control constructs, they fail to do so at a high level, that is, they fail to capture the semantics of the desired interactions. Moreover, they do not achieve a clean separation between control and interaction. Thus they cannot reconcile reusability and compliance with flexibility. In this paper, we attempt to make a clean separation between the two. Protocols are reusable as they specify only interaction (the *what*); the agents' business policies control the interaction (the *how*).

8.1 Literature

Current approaches tackle some of the above challenges, but only partially. Though service composition is a kind of reusability, it is limited to a small class of applications. The reusability we are advocating is conceptually at the level of binary libraries. In the following, we discuss some of the prevalent approaches for modeling business processes.

BPEL [1] BPEL is a flow language that is used to describe web services in terms of a process model. BPEL enables process composition by allowing the specification of partner services and using flow constructs to operationally compose the web services. In this way, BPEL takes a logically centralized view of the composed web service. Although it supports the specification of exceptions, the exceptions are hard-coded and therefore the agents' flexibility is limited.

Semantic Web Services OWL-S [4] is an ontology for web services (built using OWL, the W3C's Web Ontology Language) that enables the specification of the service profile, the service grounding, and the process model. Unlike BPEL processes that are statically composed, OWL-S processes can be dynamically composed via planning. However, the processes are specified logically centrally in a flow language. It therefore suffers from the same limitations as BPEL.

RosettaNet and ebXML RosettaNet [14] is an industry-led consortium working to create and implement industry-wide open e-business processes. RosettaNet has created more than 100 Partner Interface Processes (PIPs), which are in the nature of business protocols. RosettaNet enables the creation of business processes using PIPs, but does not directly support the creation of business processes. ebXML [5] provides a language in which specifications such as RosettaNet's can be encoded. RosettaNet and ebXML are limited to interactions with a single request-response pair.

MIT Process Handbook This effort catalogues different kinds of business processes in a hierarchy [9]. For example, *sell* is a generic business process. It can be qualified by *sell what*, *sell to who*, and so on. Our notion of a protocol hierarchy bears similarity with the Handbook, the major difference being that we attempt to give a formal semantics to the hierarchy in terms of commitments, and support aggregation in a robust manner.

Commitment Life-Cycle Fornara and Colombetti [6] present an operational characterization of commitments in which they treat commitments as objects. They model commitments as having states and, therefore, can represent a life-cycle of commitments. Fornara and Colombetti's focus is on developing semantics for an agent communication language (ACL).

The π -Calculus for Business Processes The π -calculus has recently been suggested as an approach for modeling business processes, e.g., [10]. The π -calculus can potentially be quite useful, but only if applied at the level of interaction protocols. The π -calculus is conventionally applied simply to encode orchestrations as in XLANG (now absorbed into BPEL [1]) or to even to specify choreographies as in WSCI [18]. In other words, the machinery of the π -calculus is used primarily to encode the sequence of steps to be executed—something that could be done with any conventional scripting approach. Consequently, even some proponents of the π -calculus recognize that its subtle features of the π -calculus, e.g., reconfigurability, end up not being put to good use in the current literature [17].

8.2 Directions

We have presented the elements of a formal model using the π -calculus in this paper. A current research direction is to exploit π -calculus concepts such as bisimulation to determine the *compatibility* of an agent's business policies with a role it wishes to adopt. This will lead naturally into a type system for protocols to be able to formally create a hierarchy of protocols.

On the practical side, we are developing a new language, *OWL for Protocols* or OWL-P, which can be used to create publishable specification of protocols from which role skeletons can be extracted. We are implementing this as part of a multiagent architecture which embodies the spirit of Figure 3.

References

1. BPEL. Business process execution language for web services, version 1.1, May 2003. www-106.ibm.com/developerworks/webservices/library/ws-bpel.
2. A. Chopra and M. P. Singh. Nonmonotonic commitment machines. In F. Dignum, editor, *Advances in Agent Communication: Proceedings of the 2003 AAMAS Workshop on Agent Communication Languages*, LNAI, pages 183–200. Springer-Verlag, 2003.
3. B. Cox, J. Tygar, and M. Sirbu. Netbill security and transaction protocol. In *Proceedings of the First USENIX Workshop on Electronic Commerce*, pages 77–88, 1995.
4. DAML-S. DAML-S: Web service description for the semantic Web. In *Proceedings of the 1st International Semantic Web Conference (ISWC)*, July 2002. Authored by the DAML Services Coalition, which consists of (alphabetically) Anupriya Ankolekar, Mark Burstein, Jerry R. Hobbs, Ora Lassila, David L. Martin, Drew McDermott, Sheila A. McIlraith, Srini Narayanan, Massimo Paolucci, Terry R. Payne and Katia Sycara.
5. ebXML. Electronic business using eXtensible markup language, 2002. Technical Specifications release, URL: <http://www.ebxml.org/specs/index.htm>.
6. N. Fornara and M. Colombetti. Operational specification of a commitment-based agent communication language. In *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 535–542. ACM Press, July 2002.
7. F. Guerin and J. Pitt. Denotational semantics for agent communication languages. In *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 497–504. ACM Press, 2001.
8. N. R. Jennings. Commitments and conventions: The foundation of coordination in multi-agent systems. *Knowledge Engineering Review*, 2(3):223–250, 1993.
9. T. W. Malone, K. Crowston, and G. A. Herman, editors. *Organizing Business Knowledge: The MIT Process Handbook*. MIT Press, Cambridge, MA, 2003.
10. L. G. Meredith and S. Bjorg. Contracts and types. *Communications of the ACM*, 46(10):41–47, Oct. 2003.
11. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. LFCS Technical Report 89-85, University of Edinburgh, 1989.
12. J. Parrow. An introduction to the π -calculus. In J. A. Bergstra, A. Ponse, and S. A. Smolka, editors, *Handbook of Process Algebra*, pages 479–543. Elsevier, 2001.
13. M. Robin. *Communicating and Mobile Systems: the pi-Calculus*. Cambridge University Press, 1999.
14. RosettaNet. Home page, 1998. www.rosettanet.org.
15. M. P. Singh. Agent communication languages: Rethinking the principles. *IEEE Computer*, 31(12):40–47, Dec. 1998.
16. M. P. Singh. An ontology for commitments in multiagent systems: Toward a unification of normative concepts. *Artificial Intelligence and Law*, 7:97–113, 1999.
17. L. Wischik. Process calculi for Web choreography, Mar. 2003. <http://www.wischik.com/lu/research/lucian-piforweb-w3c-mar2003-handout.pdf>.
18. WSCI. Web service choreography interface 1.0, July 2002. www.sun.com/software/xml/developers/wsci/wsci-spec-10.pdf.
19. Workflow process definition interface: XML process definition language, version 1.0, Oct. 2002. <http://www.wfmc.org/standards/docs.htm>.
20. P. Yolum and M. P. Singh. Commitment machines. In *Proceedings of the 8th International Workshop on Agent Theories, Architectures, and Languages (ATAL-01)*, pages 235–247. Springer-Verlag, 2002.