Semantic Web Service Choreography: Contracting and Enactment

Dumitru Roman[1] and Michael Kifer[2]

[1] STI Innsbruck, University of Innsbruck, Austria
[2] State University of New York at Stony Brook, USA

Abstract. The emerging paradigm of service-oriented computing requires novel techniques for various service-related tasks. Along with automated support for service discovery, selection, negotiation, and composition, support for automated service contracting and enactment is crucial for any large scale service environment, where large numbers of clients and service providers interact. Many problems in this area involve reasoning, and a number of logic-based methods to handle these problems have emerged in the field of Semantic Web Services. In this paper, we build upon our previous work where we used Concurrent Transaction Logic (CTR) to model and reason about service contracts. We significantly extend the modeling power of the previous work by allowing iterative processes in the specification of service contracts, and we extend the proof theory of CTR to enable reasoning about such contracts. With this extension, our logic-based approach is capable of modeling general services represented using languages such as WS-BPEL.

1 Introduction

The area of Semantic Web Services is one of the most promising subareas of the Semantic Web. It is the main focus of large international projects such as OWL-S,[1] SWSL,[2] WSMO,[3] DIP,[4] and SUPER,[5] which deal with service discovery, service choreography, automated contracting for services, service enactment and monitoring. In this context, the focus of this paper is modeling of behavioral aspects of services, and service contracting and enactment.

In a service-oriented environment, interaction is expected among large numbers of clients and service providers, and contracts through human interaction is out of question. To enable automatic establishment of contracts, a formal contract description language is required and a reasoning mechanism is needed to be able to verify that the contract terms are fulfilled. A contract specification has to describe the functionality of the service, values to be exchanged, procedures,

[1] http://www.daml.org/services/owl-s/
[2] http://www.w3.org/Submission/SWSF-SWSL/
[3] http://www.wsmo.org/
[4] http://dip.semanticweb.org/
[5] http://ip-super.org/

and guarantees. This paper develops such a formal framework with a particular focus on service contracting.

Our approach is based on *Concurrent Transaction Logic* (CTR) [3] and continues the line of research that looks at CTR as a unifying formalism for modeling, discovering, choreographing, contracting, and enactment of Web services [5,12,6,9]. This previous work, however, was confined to straight-line services, which cannot do repeated interactions—a serious limitation in view of the fact that the emerging languages for specifying the behavior of Web services, such as the Web Services Choreography Description Language (WS-CDL)[6] or Web Services Business Process Execution Language (WS-BPEL),[7] identify iterative processes as core elements of their models. The present paper closes this gap by extending the previous work to cover choreography of iterative processes. CTR can thus be used to model and reason about general service choreographies, including the ones definable by very expressive languages such as [13], WS-CDL, and WS-BPEL. We obtain these results by significantly extending the language for service contracts and through a corresponding extension to the proof theory of CTR. In this way, we also contribute to the body of results about CTR itself.

This paper is organized as follows. Section 2 describes the basic techniques from process modeling, such as control flow and constraints. We then review the modeling framework of [9]. Section 3 gives a short introduction to CTR to help keep the paper self-contained. Section 4 shows how the framework outlined in Section 2 is formalized in CTR. Section 5 describes the reasoning procedure—the key component of service contracting in our framework. Section 6 presents related work, and Section 7 concludes this paper.

2 Modeling, Contracting, and Enactment of Services

In [9] we identified three core service behavior aspects of service contracting: (1) *Web service choreography*—a specification of how to invoke and interact with the service in order to get results; (2) *Service policies*—a set of additional constraints imposed on the choreography and on the input; and (3) *Client contract requirements*—the contractual requirements of the user, which go beyond the basic functions (such as selling books or handling purchase orders) of the service.

The choreography of a service is described with *control* and *data flow graphs*, and service policy and clients' contract requirements are described with *constraints*. We defined the problem of *service contracting* as: given a service choreography, a set of service policies, and a set of client contract requirements, decide whether an execution of the service choreography exists, that satisfies both the service policies and the client contract requirements. Furthermore, the problem of *service enactment* was defined as finding out whether enactment of a service is possible according to a contract and, if so, finding an actual sequence of interactions that fulfils the contract.

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

[7] http://www.oasis-open.org/committees/wsbpel/

The present paper deals with the same type of problems, but the settings are significantly different: the service choreographies (control-flow graphs) are more complex since now we allow iterative interactions (whereas [9] could deal only with iteration-free interactions), and the set of allowed constraints is likewise different: it allows complex constraints on iterative interactions. We illustrate the new approach through an example of a virtual manufacturing service, which handles purchase orders (POs). Apart from the clients, the virtual manufacturing service may interact with service providers to purchase the various items requested in the purchase order. It may also contract with shippers to deliver the purchased items to the client. For each item in a PO, the virtual manufacturing service checks item availability with producers. Depending on the availability and for other reasons (e.g., the reputation of the item producers) the virtual service may choose a certain item producer-service, or it may inform the client that an item is unavailable. If an item is available, the client may choose to accept or to reject the offered item. To ship the items in the PO, the virtual service contacts shipping services. Depending on the availability of shipping services and taking a host of other considerations into account (e.g., shipper's reputation) the virtual service may book a specific shipper for delivering some or all of the items. Clients are required to provide payment guarantees. If a guarantee is provided, payment can be made for all items at once or separately for each item.

Service Choreographies. Figure 1 shows a *service choreography* depicted as a *control flow graph* of a fairly complex virtual manufacturing service. Control flow graphs are typically used to specify local execution dependencies among the interactions of the service, since they are a good way to visualize the overall flow of control. The nodes in such a graph represent *interaction tasks*, which can be thought of as functions that take inputs and produce outputs. In Figure 1, tasks are represented as labeled rectangles. The label of a rectangle is the name of the task represented by that rectangle, and a graph inside the rectangle is the *definition* or *decomposition* of that task. Such a task is called *composite.* Tasks that do not have associated graphs are *primitive.* A control flow graph can thus be viewed as a hierarchy of tasks. There is always one composite task at the root of the hierarchy. In our example, **handle_order** is the root; its subtasks include **handle_items** and **handle_shippers**. These subtasks are composite and their rectangles are shown separately (to avoid clutter). The task **place_order** is an example of a primitive task. To differentiation primitive and composite tasks, we use rectangles with gray background to depict primitive tasks.

Each composite task has an initial and the final interaction task, the successor task for each interaction task, and a sign that tells whether these successors must *all* be executed concurrently (indicated by **AND**-split nodes), or whether only one of the alternative branches needs to be executed non-deterministically (indicated by **OR**-nodes). For instance, in Figure 1(a), all successors of the initial interaction **place_order** must be executed, whereas in Figure 1(e) either **item_available** or **item_unavailable** is executed. The node **payment_guarantee** is also an **OR**-split, but with a twist. The lower branch going out of this node represent a situation where the client provides a payment.

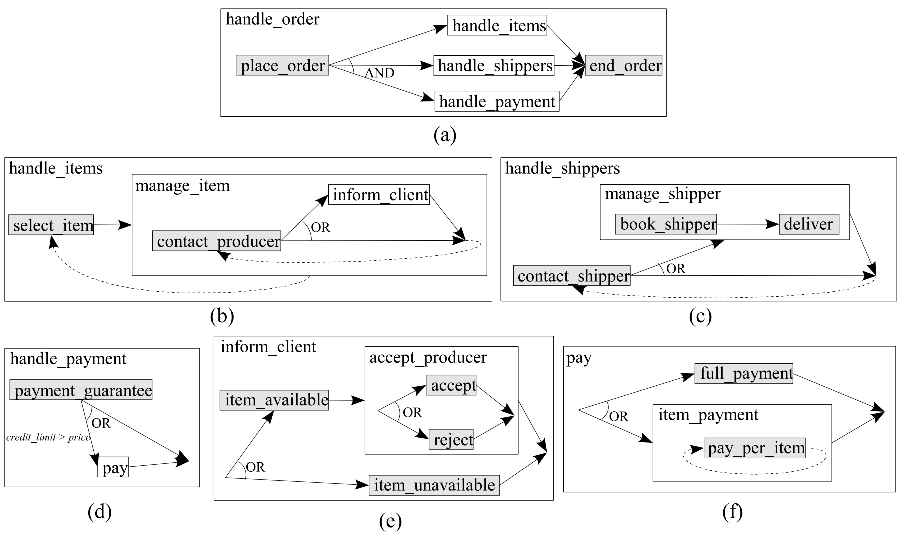

Fig. 1. A hierarchy of control-flow graphs with iterative tasks

The upper branch, has no interactions, and it joins the lower branch. This means that the lower branch is *optional*: the service may or may not accept a payment.

Control flows of some of the composite tasks in Figure 1 have dashed arrows pointing from the final task to the initial task in the respective graphs. Such arrows represent the fact that a composite task can execute iteratively multiple times. We call these *iterative* tasks and differentiate them from *non-iterative* tasks. For example, Figure 1(b) depicts **handle_items** as an iterative task where a sequence of two sub-tasks, **select_item** and **manage_item**, can be executed multiple times (for example for each item in the PO). Note also the condition *credit_limit* > *price* attached to the arc leaving the node **payment_guarantee**. Such a condition is called *transition condition.* It says that in order for the next interaction with the service to take place the condition must be satisfied. The parameters *credit_limit* and *price* may be obtained by querying the current state of the service or they may be passed as parameters from one interaction to another—the actual method depends on the concrete representation. In general, transition conditions are Boolean expressions attached to the arcs in control flow graphs. Only the arcs whose conditions are true can be followed at run time.

With these explanations, it should be clear how the control flow graph in Figure 1 represents the virtual manufacturer scenario described earlier. The top-level composite task, **handle_order** is used to process orders. Figure 1(a) depicts its decomposition: first the order is placed (**place_order**), then the items in the PO are processed (**handle_items**), delivery is arranged (**handle_shippers**), and a payment process is initiated (**handle_payment**). These three tasks are executed in parallel. Once all of them completes, the order handling is finished (**end_order**). The other parts of the figure show how each of the above subtasks are executed. The important things to observe here is that some tasks are complex and some primitive; some are to be executed in parallel (the **AND**-nodes) and some in sequence; some tasks have non-deterministic choice (the **OR**-nodes) and some are iterative (those that have dashed arcs in their definitions).

Service Policies and Client Contract Requirements. Apart from the local dependencies represented directly in control flow graphs, *global* constraints often arise as part of *policy* specification. Another case where global constraints arise is when a client has specific requirements to the interaction with the service. These requirements usually have little to do with the functionality of the service (e.g., handling orders), but rather with the particular guarantees that the client wants before entering into a contract with the service. We call such constrains *client contract requirements*. In (1) we give an example of global constraints that represent service policy and client contract requirements for our running example. Note that all the constraints in this example are on iterative tasks.

Service policy:

1. A shipper is booked only if the user accepts at least 7 items.

2. If pay per item is chosen by the user, then the payment must happen immediately before each item delivery.

3. Payment guarantee must be given before the client is informed about the availability of items.

(1)

Client contract requirements:

4. All items purchased by the client must be shipped at once.

5. If full payment is chosen by the client, then it must happen only after all purchased items are delivered.

6. Before the client purchases items the service must have booked a shipper.

Service Contracting and Enactment. With this modeling mechanism in place, we define service contracting and enactment as follows:

– *Service contracting*: given a service choreography (i.e. a hierarchical control-flow graph containing iterations), a set of service policies and client contract requirements (i.e. constraints), decide if there is an execution of the service choreography that complies both with the service policies and the client contract requirements.
– *Service enactment*: if service contracting is possible, find out an actual order of interactions that fulfils the contract, and execute it.

3 Overview of CTR

Concurrent Transaction Logic (CTR) [3] is an extension of the classical predicate logic, which allows programming and reasoning about state-changing processes. Here we summarize the relevant parts of CTR's syntax and give an informal explanation of its semantics. For details we refer the reader to [3].

Basic syntax. The atomic formulas of CTR are identical to those of the classical logic, *i.e.*, they are expressions of the form $p(t_1, \ldots, t_n)$, where p is a predicate symbol and the t_i's are function terms. More complex formulas are built with the help of connectives and quantifiers. Apart from the classical $\vee$, $\wedge$, $\neg$, $\forall$, and $\exists$, CTR has two additional connectives, $\otimes$ (*serial conjunction*) and $\mid$ (*concurrent conjunction*), and a modal operator $\odot$ (*isolated execution*). For instance, $\odot(p(X) \otimes q(X)) \mid (\forall Y(r(Y) \vee s(X,Y)))$ is a well-formed formula.

Informal semantics. Underlying the logic and its semantics is a set of database *states* and a collection of *paths*. For the purpose of this paper, the reader can think of the states as just a set of relational databases, but the logic does not rely on the exact nature of the states—it can deal with a wide variety of them.

A *path* is a finite sequence of states. For instance, if $s_1, s_2, ..., s_n$ are states, then $\langle s_1 \rangle$, $\langle s_1\ s_2 \rangle$, and $\langle s_1\ s_2\ ...\ s_n \rangle$ are paths of length 1, 2, and n, respectively.

Just as in classical logic, CTR formulas assume truth values. However, *unlike* classical logic, the truth of CTR formulas is determined over paths, *not* at states. If a formula, ϕ, is true over a path $\langle s_1, ..., s_n \rangle$, it means that ϕ can *execute* starting at state s_1. During the execution, the current state will change to s_2, s_3, ..., etc., and the execution terminates at state s_n.

With this in mind, the intended meaning of the CTR connectives can be summarized as follows:

- $\phi \otimes \psi$ *means*: execute ϕ then execute ψ. Or, model-theoretically, $\phi \otimes \psi$ is true over a path $\langle s_1, ..., s_n \rangle$ if ϕ is true over a prefix of that path, say $\langle s_1, ..., s_i \rangle$, and ψ is true over the suffix $\langle s_i, ..., s_n \rangle$.
- $\phi \mid \psi$ *means*: ϕ and ψ must both execute concurrently, in an interleaved fashion.
- $\phi \wedge \psi$ *means*: ϕ and ψ must both execute along the *same* path. In practical terms, this is best understood in terms of *constraints* on the execution. For instance, ϕ can be thought of as a transaction and ψ as a constraint on the execution of ϕ. It is this feature of the logic that lets us specify constraints as part of service contracts.
- $\phi \vee \psi$ *means*: execute ϕ *or* execute ψ non-deterministically.
- $\neg\phi$ *means*: execute in any way, provided that this will *not* be a valid execution of ϕ. Negation is an important ingredient in temporal constraint specifications.
- $\odot\phi$ *means*: execute ϕ in isolation, *i.e.*, without interleaving with other concurrently running activities. This operator enables us to specify the transactional parts of service contacts.

CTR contains a predefined propositional constant, `state`, which is true only on paths of length 1, that is, on database states. As we shall see in the next section, `state` is used in the definition of iterative tasks in service choreographies. Another propositional constant that we will use in the representation of constraints is `path`, defined as `state` $\vee$ $\neg$`state`, which is true on every path.

Concurrent-Horn subset of CTR. Implication $p \longleftarrow q$ is defined as $p \vee \neg q$. The form and the purpose of the implication in CTR is similar to that of Datalog: p can be thought of as the name of a procedure and q as the definition of that procedure. However, unlike Datalog, both p and q assume truth values on execution paths, not at states.

More precisely, $p \longleftarrow q$ means: if q can execute along a path $\langle s_1, ..., s_n \rangle$, then so can p. If p is viewed as a subroutine name, then the meaning can be re-phrased as: one way to execute p is to execute its definition, q.

The control flow parts of service choreographies are formally represented as *concurrent-Horn goals* and *concurrent Horn rules*. A *concurrent Horn goal* is:

- any atomic formula is a concurrent-Horn goal;
- $\phi \otimes \psi$, $\phi \mid \psi$, and $\phi \vee \psi$ are concurrent-Horn goals, if so are ϕ and ψ;
- $\odot\phi$ is a concurrent-Horn goals, if so is ϕ.

A *concurrent-Horn rule* is a CTR formula of the form $head \longleftarrow body$, where *head* is an atomic formula and *body* is a concurrent-Horn goal. The concurrent-Horn fragment of CTR is efficiently implementable, and there is an SLD-style proof procedure that proves concurrent-Horn formulas and *executes* them at the same time [3]. Observe that the definition of concurrent-Horn rules and goals *does not* include the connective $\wedge$. In general, $\wedge$ represents *constrained execution.* The present work deals with a much larger class of constraints than [5,9], which includes iterative processes, and formulas of the form $ConcurrentHornGoal \wedge Constraints$ are handled by the extended proof theory in Section 5.

Elementary updates. In CTR *elementary updates* are formulas that change the underlying database state. Semantically they are binary relations over states. For instance, if $\langle s_1 \ s_2 \rangle$ belongs to the relation corresponding to an elementary update u, it means that u can cause a transition from state s_1 to state s_2.

Constraints. Because transactions are defined on paths, CTR can express a wide variety of constraints on the way transactions execute. One can place conditions on the state of the database during transaction execution (constraints based on serial conjunction), or may forbid certain sequences of states (constraints based on serial implication). To express the former, the proposition constant `path`, introduced above, is used; for example, $\mathtt{path} \otimes \psi \otimes \mathtt{path}$ specifies a path on which ψ must be true. To express the latter, the binary connectives "$\Leftarrow$" and "$\Rightarrow$" are used. The formula $\psi \Leftarrow \phi$ means that whenever ψ occurs, then ϕ occurs right after it. The formula $\psi \Rightarrow \phi$ means that whenever ϕ occurs, then ϕ must have occurred just before it.

4 Formalizing Service Contracts

We begin our formalization by showing how service choreographies are represented in CTR (Section 4.1). Section 4.2 proceeds to formalize service policies and contract requirements as constraints expressed in CTR. Section 4.3 provides a discussion on the assumption we take when modeling service contracts.

4.1 Modeling Service Choreography with CTR

In CTR, *tasks* are represented as formulas of the form $p(X_1, \ldots , X_n)$, where p is a predicate symbol and the X_i's are variables. The predicate symbol is the name of the task, and the variables are placeholders for data items that the task manipulates (e.g. inputs, outputs, etc.). A *task instance* is a task whose variables are substituted with concrete values.

Definition 1. (Dependency between tasks) *A task p_1* depends *on a task p_2 if p_2 appears in the body of a rule that has p_1 as its head.*

Definition 2. (Primitive task) *A task is* primitive *if it does not depend on any other task.*

We conceptualize primitive tasks as opaque actions that produce some external action. In CTR, such actions are represented as elementary updates and so ground instances of primitive tasks are treated as CTR's elementary updates.

Definition 3. (Non-iterative task) *A* non-iterative *composite task, p, is a task defined by a rule of the form $p \leftarrow q$, where q is a CTR goal none of whose tasks depends on p.*

Definition 4. (Iterative task) *An* iterative *task, p, is a task defined by a rule of the form $p \leftarrow (q \otimes p) \vee \texttt{state}$, where q is a CTR goal none of whose tasks depends on p. This is equivalent to a pair of concurrent-Horn rules.*

Definition 5. (Service choreography) *A* service choreography *is an iterative or non-iterative composite task that represents the root of the task hierarchy, along with the rules defining it.*

A CTR representation of the service choreography from Figure 1 is shown in (2), below. The *handle_order* task is the root of task hierarchy; it is followed by the rules that define it.

$$
\begin{array}{l}
handle\_order \leftarrow place\_order \otimes \\
\qquad\qquad (handle\_items \mid handle\_shippers \mid handle\_payment) \otimes end\_order \\
handle\_items \leftarrow (select\_item \otimes manage\_item \otimes handle\_items) \vee \texttt{state} \\
manage\_item \leftarrow (contact\_producer \otimes (inform\_client \vee state) \otimes manage\_item) \vee \texttt{state} \\
handle\_shippers \leftarrow (contact\_shipper \otimes (manage\_shipper \vee \texttt{state}) \otimes handle\_shippers) \vee \texttt{state} \\
manage\_shipper \leftarrow book\_shipper \otimes deliver \\
handle\_payment \leftarrow payment\_guarantee \otimes ((credit\_limit > price \otimes pay) \vee \texttt{state}) \\
inform\_client \leftarrow (item\_available \otimes accept\_producer) \vee item\_unavailable \\
accept\_producer \leftarrow accept \vee reject \\
pay \leftarrow full\_payment \vee item\_payment \\
item\_payment \leftarrow (pay\_per\_item \otimes item\_payment) \vee \texttt{state}
\end{array}
\tag{2}
$$

4.2 Modeling Constraints Using CTR

We now define an algebra of constraints, $\mathcal{C}onstr$, which we will use in this paper to model service policies.

Definition 6. (Constraints) *The following constraints form the algebra $\mathcal{C}onstr$ (and nothing else):*

1. **Primitive constraints***: If a is a task in a service choreography, then the following are* ***primitive*** *constraints:*
   - *existence*(a, n)—task a must execute at least n times ($n \geq 1$)*:* $\nabla_{\geq n} a \equiv \underbrace{\nabla_{\geq 1} a \otimes ... \otimes \nabla_{\geq 1} a}_{n \geq 1}$*, where* $\nabla_{\geq 1} a \equiv \texttt{path} \otimes a \otimes \texttt{path}$. [8]

[8] We will also use ∇a as an abbreviation for $\nabla_{\geq 1} a$ as this constraint occurs very frequently.

- $absence(a)$ - task a must not execute*:* $\neg\nabla a \equiv \neg(\mathtt{path} \otimes a \otimes \mathtt{path})$
- $exactly(a,n)$ - task a must execute exactly n times (n $\geq$ 1)*:* $\nabla_n a \equiv \underbrace{\nabla_1 a \otimes ... \otimes \nabla_1 a}_{n\geq 1}$*, where* $\nabla_1 a \equiv \neg\nabla a \otimes a \otimes \neg\nabla a$*.*

2. **Serial constraints***: If a, b are tasks in a service choreography then the following are* ***serial*** *constraints:*
   - $after(a,b)$ - whenever a executes, b has to be executed after it. Task b does not have to execute immediately after a, and several other instances of a might execute before b does*:* $(\mathtt{path} \otimes a) \Rightarrow \nabla b$
   - $before(a,b)$ - whenever b executes, it must be preceded by an execution of a. Task a does not have to execute immediately before b: $\nabla a \Leftarrow (b \otimes \mathtt{path})$
   - $blocks(a,b)$ - if a executes, b can no longer be executed in the future: $(\mathtt{path} \otimes a) \Rightarrow \neg\nabla b$
   - $between(a,b,a)$ - b must execute between any two executions of a, i.e. after an execution of a, any subsequent execution of a is blocked until b is executed: $(\mathtt{path} \otimes a) \Rightarrow \nabla b \Leftarrow (a \otimes \mathtt{path})$
   - $not\text{-}between(a,b,a)$ - b must not execute between any pair of executions of a. If b executes after a, no future execution of a is possible: $(\mathtt{path} \otimes a) \Rightarrow \neg\nabla b \Leftarrow (a \otimes \mathtt{path})$
3. **Immediate serial constraints***: If a, b are tasks in a service choreography then the following are* ***immediate serial*** *constraints:*
   - $right\text{-}after(a,b)$ - whenever a executes, b has to execute immediately after it*:* $(\mathtt{path} \otimes a) \Rightarrow (b \otimes \mathtt{path})$
   - $right\text{-}before(a,b)$ - whenever b executes, a has to be executed immediately before it*:* $(\mathtt{path} \otimes a) \Leftarrow (b \otimes \mathtt{path})$
   - $not\text{-}right\text{-}after(a,b)$ - whenever a and b execute, b must not execute immediately after a, i.e. between the execution of a and b there must be an execution of a task other than a and b*:* $(\mathtt{path} \otimes a) \Rightarrow (\neg\mathtt{state} \wedge \neg\nabla a \wedge \neg\nabla b) \Leftarrow (b \otimes \mathtt{path})$

   The negation of $right\text{-}before(a,b)$ *is equivalent to* $not\text{-}right\text{-}after(b,a)$*, so we do not define it explicitly.*

4. **Complex constraints***:If* C_1, $C_2 \in \mathcal{C}\textsc{onstr}$ *then so are* $C_1 \wedge C_2$*, and* $C_1 \vee C_2$*.*

The following examples illustrate the diverse set of constraints that can be expressed with the help of $\mathcal{C}\textsc{onstr}$.

- $\neg\nabla a \vee \nabla_1 a \vee \nabla_2 a \vee ... \vee \nabla_n a$ — task a must execute at most n times. We denote this constraint by *at-most(a,n)*. This constraint together with the primitive constraints introduced earlier capture the set of *existence formulas* from [13].
- $\neg\nabla a \vee \nabla b$ — if a is executed, then b must also execute (before or after a).
- $(\neg\nabla a \vee \nabla b) \wedge (\neg\nabla b \vee \nabla a)$ — if a is executed, then b must also be executed, and vice versa.
- $after(a,b) \wedge before(a,b)$ — every occurrence of task a must be followed by an occurrence of task b and there must be an occurrence of a before every occurrence of b.
- $\neg\nabla a \vee between(a,b,a)$ — if task a is executed then b must execute after it, and before that b there can be no other a.
- $\neg\nabla b \vee (before(a,b) \wedge between(b,a,b))$ — if task b is executed, it has to be preceded by an occurrence of a. The next instance of b can executed only after another occurrence a.
- $between(a,b,a) \wedge between(b,a,b)$ — tasks a and b must alternate.

– $right\text{-}after(a, b) \wedge right\text{-}before(a, b)$ — executions of a and b must be next to each other with no intervening tasks in-between.
– $\neg\nabla a \vee \neg\nabla b$ — it is not possible for a and b to execute in the same choreography run.
– $not\text{-}between(a, b, a) \wedge not\text{-}between(b, a, b)$ — b must not execute between any two executions of a, and a must not execute between any two executions of b.

With the modeling mechanism in place, we can now represent the constraints from (1) formally:

$$\begin{array}{ll}
1.\ (at\text{-}most(accept, 6) \wedge absence(book\_shipper)) \vee & \\
\quad (existence(accept, 7) \wedge after(accept, book\_shipper)) & \\
2.\ absence(item\_payment) \vee right\_before(pay\_per\_item, deliver) & \\
3.\ before(payment\_guarantee, inform\_client) & (2)\\
4.\ exactly(deliver, 1) & \\
5.\ absence(full\_payment) \vee (before(deliver, full\_payment) \wedge block(full\_payment, deliver)) & \\
6.\ before(book\_shipper, pay) &
\end{array}$$

4.3 Service Contracts Assumption

We now introduce the assumptions about the forms of the constraints and tasks involved in service choreography. These assumptions do not limit the modeling power of the language in the sense that any service choreography can be simulated by another choreography that satisfies these assumptions.

Primitive Tasks Independence Assumption. A service choreography, G, satisfies the independence assumption iff all of its primitive tasks are independent of each other; two primitive tasks are said to be *independent* iff they are represented by *disjoint* binary relations over database states.[9] This assumption means that a transition between two states is caused by precisely one primitive task, and no other task can cause the transition between those states. It is easy to see that the independence assumption does not limit the modeling power in the following sense: there is a 1-1 correspondence between executions of the original choreography and executions of the instrumented choreography.

Constraints Based on Primitive Tasks. Our service contracting reasoning technique (developed in Section 5) assumes that constraints are based on primitive tasks: a set of constraints, C, is said to be based on primitive tasks iff all tasks appearing in C are primitive tasks. As with the independence assumption, the above restriction on constraints does not limit the modeling power of the language. It is easy to instrument composite tasks in such a way that constraints that the resulting set of constraints will be based on primitive tasks only. More specifically, every composite task, p, can be changed as follows: $p_{start} \otimes p \otimes p_{end}$, where p_{start} and p_{end} are new *unique* primitive tasks. The effect is that now each composite task has a clearly identified begin- and end-subtask, which can be used in constraints. For instance, the constraint *between(a,b,a)* is now equivalent to *between*$(a,b_{start},a) \wedge$ *between*(a,b_{end},a). We can also have constraints such as *before*(a_{start},b_{end}) and *between*$(a_{start},b_{start},a_{end})$.

Unique Task Occurrence Assumption. Some of our results depend on the *unique task occurrence assumption*, which informally says that each task can

[9] Recall that primitive tasks are represented by elementary updates of CTR, and an elementary update is a binary relation over database states.

occur only once in the conjunctive part of the definition of any composite task. The unique task occurrence assumption does not limit the modeling power of our language, since the different occurrences of such tasks can be renamed apart.

Definition 7. (Service Contracts Assumption) *A service choreography G and a set of constraints C satisfy the* service contract assumption *iff the primitive tasks of G satisfy the independence assumption and G has the unique task occurrence property. In addition, the set of constraints C must be based on primitive tasks.*

5 Reasoning about Contracts

Let C be a constraint from $\mathcal{C}_{\mathcal{ONSTR}}$, which includes the service policy and the client contract requirements. Let G be a a service choreography. Suppose G and C satisfy the service contracts assumption. Then

1. **Contracting**: The problem of determining if contracting for the service is possible is finding out if an execution of the CTR formula $G \wedge C$ exists.
2. **Enactment**: The problem of enactment is formally defined as finding a constructive *proof* for formulas of the form $G \wedge C$. A constructive proof is a sequence of inference rules of CTR that starts with an axiom and end with the formula $G \wedge C$. Each such proof gives us a way to execute the choreography so that all constraints are satisfied.

The rest of this section develops a proof theory for formulas of the form $G \wedge C$, where G is a service choreography and C is a constraint in $\mathcal{C}_{\mathcal{ONSTR}}$. Section 5.1 presents a simplification operation used by the extended proof theory, and Section 5.2 presents the actual proof theory.

5.1 The Simplification Transformation

First, we define an auxiliary simplification transformation, $\mathcal{S}$. If G is a choreography and σ a primitive constraint, then $\mathcal{S}(G, \sigma)$ is also a service choreography (in particular, it does not contain the logical connective $\wedge$). If G has the unique task occurrence property then $\mathcal{S}(G, \sigma)$ is defined in such a way that the following is true: $\mathcal{S}(G, \sigma) \equiv G \wedge \sigma$. In other words, $\mathcal{S}$ is a transformation that eliminates the primitive constraint σ from $G \wedge \sigma$ by "compiling" it into the service choreography.[10] The following defines the simplification transformation $\mathcal{S}$.

Definition 8. (Simplification transformation) Let s be a primitive task from $\mathcal{T}_{\mathcal{ASKS}}$. Let t be a another primitive task from $\mathcal{T}_{\mathcal{ASKS}}$. Then*:*

$\mathcal{S}(t, \nabla t) = t$; $\mathcal{S}(t, \nabla_{\geq n} t) = \neg\mathtt{path}$; $\mathcal{S}(t, \neg\nabla t) = \neg\mathtt{path}$; $\mathcal{S}(t, \nabla_1 t) = t$;
$\mathcal{S}(t, \nabla_n t) = \neg\mathtt{path}$; $\mathcal{S}(t, \nabla s) = \neg\mathtt{path}$; $\mathcal{S}(t, \nabla_{\geq n} s) = \neg\mathtt{path}$;
$\mathcal{S}(t, \neg\nabla s) = t$; $\mathcal{S}(t, \nabla_1 s) = \neg\mathtt{path}$; $\mathcal{S}(t, \nabla_n s) = \neg\mathtt{path}$;

[10] Note that the conjunction $G \wedge \sigma$ can be an inconsistency.

We remind that $\neg\texttt{path}$ means inconsistency so if a conjunct reduces to $\neg\texttt{path}$ then the whole conjunction is inconsistent and if a disjunct is found to be inconsistent then it can be eliminated.

Let $p \in \mathcal{T}_{\mathcal{ASKS}}$ be an *iterative* task of the form (4) (i.e. $p \leftarrow (q \otimes p) \vee \texttt{state}$) that satisfies the unique task assumption. Then:

$$\begin{aligned}
&\mathcal{S}(p, \triangledown_{\geq n} s) =_{n=k_1+\ldots+k_m}^{\bigvee} \; (p \otimes \mathcal{S}(q, \triangledown_{\geq k_1} s) \otimes p \;\; \otimes \mathcal{S}(q, \triangledown_{\geq k_2} s) \otimes p \otimes \\
&\qquad\qquad\qquad\qquad\qquad\qquad \ldots \;\; \otimes \mathcal{S}(q, \triangledown_{\geq k_m} s) \otimes p) \\
&\mathcal{S}(p, \neg\triangledown s) = p', \text{ where } p' \text{ is defined as: } p' \leftarrow (\mathcal{S}(q, \neg\triangledown s) \otimes p') \vee \texttt{state} \\
&\mathcal{S}(p, \triangledown_n s) =_{n=k_1+\ldots+k_m}^{\bigvee} \; (\mathcal{S}(p, \neg\triangledown s) \otimes \mathcal{S}(q, \triangledown_{k_1} s) \;\; \otimes \mathcal{S}(p, \neg\triangledown s) \otimes \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathcal{S}(q, \triangledown_{k_2} s) \otimes \mathcal{S}(p, \neg\triangledown s) \otimes \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \ldots \qquad \ldots \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathcal{S}(q, \triangledown_{k_m} s) \otimes \mathcal{S}(p, \neg\triangledown s))
\end{aligned}$$

Let $r \in \mathcal{T}_{\mathcal{ASKS}}$ be a composite *non-iterative* task of the form (3) (i.e. $r \leftarrow q$) that satisfies the unique task assumption. Let δ stand for $\neg\triangledown s$, $\triangledown_{\geq n} s$, or $\triangledown_n s$, where $n \geq 1$. Then, $\mathcal{S}(r, \delta) = \mathcal{S}(q, \delta)$. Since q can have the forms $u \otimes v$, $u \mid v$, $\odot u$, or $u \vee v$, $\mathcal{S}(q, \delta)$ is obtained as follows:

$$\begin{aligned}
\mathcal{S}(u \otimes v, \delta) &= \begin{cases} (\mathcal{S}(u, \delta) \otimes v) \vee (u \otimes \mathcal{S}(v, \delta)), & \textit{if } \delta \textit{ is } \triangledown_{\geq n} s \textit{ or } \triangledown_n s \\ \mathcal{S}(u, \delta) \otimes \mathcal{S}(u \otimes v, \delta), & \textit{if } \delta \textit{ is } \neg\triangledown s \end{cases} \\
\mathcal{S}(u \mid v, \delta) &= \begin{cases} (\mathcal{S}(u, \delta) \mid v) \vee (u \mid \mathcal{S}(v, \delta)), & \textit{if } \delta \textit{ is } \triangledown_{\geq n} s \textit{ or } \triangledown_n s \\ \mathcal{S}(u, \delta) \mid \mathcal{S}(v, \delta), & \textit{if } \delta \textit{ is } \neg\triangledown s \end{cases} \\
\mathcal{S}(\odot u, \delta) &= \odot \mathcal{S}(u, \delta) \\
\mathcal{S}((u \vee v), \delta) &= \mathcal{S}(u, \delta) \vee \mathcal{S}(v, \delta)
\end{aligned}$$

5.2 Extended Proof Theory

This section develops a proof theory for formulas of the form $G \wedge C$, where G is a service choreography and $C \in \mathcal{C}_{\mathcal{ONSTR}}$.

It is easy to see that C is equivalent to $\vee_i(\wedge_j \; C_{ij})$, where each C_{ij} is either a primitive or serial constraint. To check if there is an execution of $\psi \wedge C$, we need to use the inference rules introduced below and apply them to each disjunct $\psi \wedge (\wedge_j \; C_{ij})$ separately. Therefore, we can assume that our constraint C is a set of primitive or serial constraints.

Hot Components. We remind the notion of *hot components* of a formula from [3]: *hot*(ψ) is a set of subformulas of ψ which are "ready to be executed." This set is defined inductively as follows:

1. hot(()) = {}, where () is the empty goal
2. hot(ψ) = ψ, if ψ is an atomic formula
3. hot($\psi_1 \otimes ... \otimes \psi_n$) = hot($\psi_1$)
4. hot($\psi_1 \mid ... \mid \psi_n$) = hot($\psi_1$) $\cup ... \cup$ hot(ψ_n)
5. hot($\odot\psi$) ={$\odot\psi$}
6. hot($\psi_1 \vee ... \vee \psi_n$) = hot($\psi_1$) or ... or hot($\psi_n$)

Eligible Components. The set of *eligible* components is used in deciding which inference rules are applicable at any given moment in a proof. Let ψ be a service choreography and C a set of constraints. Let $tasks(C)$ denote the set of all tasks mentioned by the constraints in C. The set of eligible components of a CTR goal ψ with respect to a set of constraints C is initially defined as follows:

$$\begin{aligned} eligible(\psi, C) = \{t \mid t \in hot(\psi),\ & \text{and } C \text{ has no constraints of the form} \\ & before(X,t) \text{ or } right\text{-}before(X,t), \\ & \text{where } X \in tasks(C) \ or\ X = ?\} \end{aligned}$$

The *eligible* set keeps changing as the tasks in the choreography execute. The exact mechanism of these changes is described in the inference rule 4. Note the use of the "?" symbol in the definition of eligible: it appears in constraints of the form *before(?,t)*, which are added or deleted during the execution, by inference rule 4. The constraint $before(?, t)$ means that for t to execute, one task (which is different from t), denoted by "?", must execute prior to t. The symbol "?" also occurs as part of a new kind of constraints which are used internally by the proof procedure: $right\text{-}before^{+}(a, b) \overset{def}{=} ? \otimes right\text{-}before(a, b)$.

This constraint means that the first task can be anything (denoted by "?"), but beginning with the second action the constraint $right\text{-}before(a, b)$ must hold during the rest of the execution. Such constraints are not present initially, but they are introduced by the proof theory system.

Sequents. Let P be a set of composite task definitions. The proof theory manipulates expressions of the form $\mathrm{P}, \mathrm{D}\text{---}\vdash (\exists)\,\phi$, called *sequents*, where P is a set of task definitions and D is the underlying database state. Informally, a sequent is a statement that the transaction $(\exists)\,\phi$, which is defined by the rules in P, can execute starting at state D. Each inference rule has two sequents, one above the other, which is interpreted as: If the upper sequent is inferred, then the lower sequent should also be inferred. As in classical resolution, any instance of an answer-substitution is a valid answer to a query.

This inference system extends the system for Horn CTR given in [3] with one additional inference rule (rule 3). The other rules from [3] are also significantly modified. The new system reduces to the old one when the set C of constraints is empty. The new system also extends the proof theory developed in [9].

Axioms. $P,\ D$ --- $\vdash () \wedge C$, for any database state D, where C does not contain constraints of the form $\nabla_{\geq n} s$ or $\nabla_n s$, where $n \geq 1$.

Inference Rules. In rules 1-5 below, σ denotes a substitution, ψ and ψ' are service choreographies, C and C' are constraint sets, D, D_1, D_2 denote database states, and a is an atomic formula in *eligible(ψ)*.

1. *Applying transaction definitions*: Let $b \leftarrow \beta$ be a rule in P, and assume that its variables have been renamed so that none are shared with ψ. If a and b unify with the most general unifier σ then

$$\frac{P,\, D \text{ ---}\vdash (\exists)\ (\psi' \wedge C)\ \sigma}{P,\, D \text{ ---}\vdash (\exists)\ \psi \wedge C}$$

where ψ' is obtained from ψ by replacing an eligible occurrence of a by β.

2. *Querying the database*: If $(\exists)a\sigma$ is true in the current state D and $a\sigma$ and $\psi'\sigma$ share no variables then

$$\frac{P,\, D \text{ ---}\vdash (\exists)\ (\psi' \wedge C)\ \sigma}{P,\, D \text{ ---}\vdash (\exists)\ \psi \wedge C}$$

where ψ' is obtained from ψ by deleting an eligible occurrence of a.

3. *Simplification*: If δ is a primitive constraint, then

$$\frac{P,\, D \text{ ---}\vdash (\exists)\ (\mathcal{S}(\psi,\delta) \wedge C)}{P,\, D \text{ ---}\vdash (\exists)\ \psi \wedge (C \wedge \delta)}$$

4. *Execution of primitive tasks*: If $a\sigma$ is a primitive task that changes state D_1 to D_2 then

$$\frac{P,\, D_2 \text{ ---}\vdash (\exists)\ (\psi' \wedge C')\ \sigma}{P,\, D_1 \text{ ---}\vdash (\exists)\ \psi \wedge C}$$

where ψ' is obtained from ψ by deleting an eligible occurrence of a. C' is obtained from C as follows. Suppose $T, S \in tasks(C)$ are arbitrary task names, and that $T \neq a$. Then:

– Step 1: Initially C' is C.
– Step 2:
  (a) replace every constraint of the form $right\text{-}before^{+}(T, S)$ in C' with a constraint of the form $right\text{-}before(T, S)$
  (b) delete every constraint of the form $before(a, T)$ in C'
  (c) delete every constraint of the form $before(?, T)$ in C'
  (d) replace every constraint of the form $right\text{-}before(a, T)$ in C' with a constraint of the form $right\text{-}before^{+}(a, T)$
  (e) for every constraint of the form $not\text{-}between(a, T, a)$ in C', add a constraint of the form $blocks(T, a)$ to C'
  (f) for every constraint of the form $after(a, T)$ in C', add a constraint of the form $existence(T)$ to C'
  (g) for every constraint of the form $blocks(a, T)$ in C', add a constraint of the form $absence(T)$ to C'
  (h) for every constraint of the form $right\text{-}after(a, T)$ in C', for all S in $tasks(C)$, $S \neq T$, add $before(T, S)$ to C'
  (i) for every constraint of the form $not\text{-}right\text{-}after(a, T)$ in C', add a constraint of the form $before(?, T)$ to C'
  (j) for every constraint of the form $between(a, T, a)$ in C', add a constraint of the form $before(T, a)$ to C'

5. *Execution of atomic transactions*: If $\odot\alpha$ is a hot component in ψ then

$$\frac{P, D \;\text{---}\!\vdash (\exists)\ (\alpha \otimes\ \psi') \wedge C}{P, D \;\text{---}\!\vdash (\exists)\ \psi \wedge C}$$

where ψ' is obtained from ψ by deleting an eligible occurrence of $\odot\alpha$.

Theorem 1. *The above inference system is sound and complete for proving constraint service choreographies, if the service choreographies and constraints satisfy the service contracts assumption.*

Proof: The proof is given in the technical report [10].

6 Related Work

The closest to the present paper is our earlier work [9]. By allowing iterative processes in choreography descriptions, as well as new kind of constraints that can be applied to iterative processes, the current paper significantly extends the modeling power of the service contract framework developed in [9]. The extension of the reasoning mechanism developed in this paper is also quite different and more general than the approach taken in [9].

DecSerFlow [13] is a service flow language that is closely related to our service behavior modeling framework. It uses Linear Temporal Logic to formalize service flows and automata theory to enact service specifications. The relations between tasks are entirely described in terms of constraints. First, our constraint algebra $\mathcal{C}$ONSTR is more expressive than DecSerFlow. Second, by combining constraints with control-flow graphs, our framework appears closer to the current practices in workflow modeling. Third, data flow and conditional control flow are easily available in our framework [9], while, to the best of our knowledge, they have not been developed in the context of DecSerFlow.

An emerging area related to our work is that of compliance checking between business processes and business contracts. For example, in [7,8] both processes and contracts are represented in a formal contract language called FCL. FCL is based on a formalism for the representation of contrary-to-duty obligations, i.e., obligations that take place when other obligations are violated as typically applied to penalties in contracts. Based on this, the authors give a semantic definition for compliance, but no practical algorithms. In contrast, our work provides a proof theory for the logic we use for service contracting.

Several other works, although not directly related to our approach to service contracting and enactment, are relevant [11,1,2,4]. Most of them present logical languages for representing contracts in various contexts. However, they are mainly based on normative deontic notions of obligation, prohibition, and permission, and thus could be seen as complementary to our approach.

In process modeling, the main other tools are Petri nets, process algebras, and temporal logic. The advantage of CTR over these approaches is that it is a *unifying* formalism that integrates a number of process modeling paradigms ranging from conditional control flows to data flows to hierarchical modeling to constraints, and even to game-theoretic aspects of multiagent processes (see, for example, [6]). Moreover, CTR models the various aspects of processes in distinct ways, which enabled us to devise algorithms with better complexity than the previously known techniques from the area of model checking for temporal logic specifications [5].

7 Conclusions

We have extended the CTR-based logic language for specifying Web service choreography and contracts to include iterative processes. As mentioned in the introduction, many practical languages for describing service behavior include iterative processes in their models and enabling reasoning about iterative processes opens up new possibilities for automated service contracting and enactment on top of existing behavioral languages. In this way, we have closed most of the outstanding problems in logic-based process modeling, which were raised in [5]. We have also extended the proof theory of CTR and made it capable of handling complex practical problems in process modeling and enactment. Due to space limitation, we did not include such modeling aspects as dataflow and conditional control flow, but this can be handled similarly to [9].

Some problems still remain. For instance, reasoning about dynamically created multiple instances of subprocesses is largely future work.

Acknowledgments. Part of this work was done while Michael Kifer was visiting Free University of Bozen-Bolzano, Italy. His work was partially supported by the BIT Institute, NSF grant IIS-0534419, and US Army Research Office under a subcontract from BNL. Dumitru Roman was partly funded by the BIT Institute, and the projects SUPER (FP6-026850), Knowledge Web (FP6-507482), SWING (FP6-26514), SHAPE (ICT-2007-216408), and EASTWEB (Contract TH/Asia Link/010 (111084)).

References

1. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Montali, M., Torroni, P.: Expressing and verifying business contracts with abductive logic programming. Number 07122 in Dagstuhl Seminar Proceedings (2007)
2. Andersen, J., Elsborg, E., Henglein, F., Simonsen, J.G., Stefansen, C.: Compositional specification of commercial contracts. Int. J. Softw. Tools Technol. Transf. 8(6), 485–516 (2006)
3. Bonner, A.J., Kifer, M.: Concurrency and Communication in Transaction Logic. In: Joint International Conference and Symposium on Logic Programming (1996)
4. Carpineti, S., Castagna, G., Laneve, C., Padovani, L.: A formal account of contracts for web services. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 148–162. Springer, Heidelberg (2006)
5. Davulcu, H., Kifer, M., Ramakrishnan, C.R., Ramakrishnan, I.V.: Logic Based Modeling and Analysis of Workflows. In: PODS, pp. 25–33 (1998)

6. Davulcu, H., Kifer, M., Ramakrishnan, I.: CTR–S: A Logic for Specifying Contracts in Semantic Web Services. In: WWW 2004, p. 144 (2004)
7. Governatori, G., Milosevic, Z., Sadiq, S.: Compliance checking between business processes and business contracts. In: EDOC 2006, pp. 221–232 (2006)
8. Governatori, G., Milosevic, Z., Sadiq, S., Orlowska, M.: On compliance of business processes with business contracts. Technical report, File System Repository (Australia) (2007), `http://search.arrow.edu.au/apps/ArrowUI/OAIHandler`
9. Roman, D., Kifer, M.: Reasoning about the behavior of semantic web services with concurrent transaction logic. In: VLDB, pp. 627–638 (2007)
10. Roman, D., Kifer, M.: Service contracting: A logic-based approach. Tech. report (2008), `http://www.wsmo.org/TR/d14/ServiceContracting140508.pdf`
11. Angelov, P.G.S.: B2B E-Contracting: A Survey of Existing Projects and Standards. Report I/RS/2003/119, Telematica Instituut (2003)
12. Senkul, P., Kifer, M., Toroslu, I.: A Logical Framework for Scheduling Workflows under Resource Allocation Constraints. In: VLDB 2002, pp. 694–705 (2002)
13. van der Aalst, W., Pesic, M.: DecSerFlow: Towards a Truly Declarative Service Flow Language. In: The Role of Business Processes in Service Oriented Architectures, number 06291 in Dagstuhl Seminar Proceedings (2006)