WSMO Choreography: From Abstract State Machines to Concurrent Transaction Logic*

Dumitru Roman[1], Michael Kifer[2], and Dieter Fensel[1]

[1] STI Innsbruck, Austria
[2] Stony Brook University, USA

Abstract. Several approaches to semantic Web services, including OWL-S, SWSF, and WSMO, have been proposed in the literature with the aim to enable automation of various tasks related to Web services, including discovery, contracting, enactment, monitoring, and mediation. The ability to specify processes and to reason about them is central to these initiatives. In this paper we analyze the WSMO choreography model, which is based on Abstract State Machines (ASMs), and propose a methodology for generating WSMO choreography from visual specifications. We point out the limitations of the current WSMO model and propose a faithful extension that is based on Concurrent Transaction Logic (CTR). The advantage of a CTR-based model is that it uniformly captures a number of aspects that previously required separate mechanisms or were not captured at all. These include process specification, contracting for services, service enactment, and reasoning.

1 Introduction

The field of Semantic Web services marries the technology of delivering services over the Web with the Semantic Web. This brings up a variety of issues, which range from service-specific and domain ontologies to service discovery, service choreography (i.e., specification of how autonomous client agents interact with services), automated contracting for services, service enactment, execution monitoring, and others. A number of past and ongoing projects proposed solutions to some of these problems. These include OWL-S,[3] SWSL[4] WSMO,[5] and WS-CDL.[6] WSMO is one of the more comprehensive approaches to Semantic Web Services, as it covers virtually all of the aforementioned areas. In this paper, we focus on one particular aspect of WSMO—its support for service choreography, which is based on the formalism of Abstract State Machines (ASMs) [3].

* Proceedings of the European Semantic Web Conference (ESWC) 2008. Springer LNCS 5021, pp. 659-673.

[3] http://www.daml.org/services/owl-s/
[4] http://www.w3.org/Submission/SWSF-SWSL/
[5] http://www.wsmo.org/
[6] http://www.w3.org/TR/ws-cdl-10/

Although ASMs have been used to design and verify software in the past, this formalism is too general and leaves Web service designers to their own devices. To the best of our knowledge, there has been little work to help guide Web service designers through the process of actual building of ASM-based choreographies. Our first contribution is to show that control flow graphs, which are commonly used to design workflows (and thus can be used as visual design tools for choreographies), have a modular translation into ASMs and can be used to specify WSMO choreorgraphy interfaces.

Next, we discuss some of the drawbacks of using ASMs in WSMO. We argue that the problem of choreography is inextricably related to the problem of contracting for Web services. We define one very useful instance of the problem of contracting and show that ASMs are inadequate to deal with this problem. We then propose a solution grounded in Concurrent Transaction Logic (CTR) [2]. Based on our previous results [15], we show that both service choreography and service contracts have natural representation in CTR: the former as deductive rules and the latter as constraints. In addition, CTR enables us to reason about consistency of the contracts with the choreographies, about service enactment, and other issues. In this way, CTR-based WSMO choreography can be seen as a natural extension of the ASM-based WSMO choreography.

An introduction to WSMO choreography is given in Section 2. Section 3 proposes a more intuitive, visual language for specifying service interactions, and develops a methodology for automatic generation of WSMO choreography transition rules. Section 4 defines the problem of service contracting as choreography under the constraints specified as service policies and customers requirements and highlights the limitations of the current WSMO choreography in this area. It then proposes CTR as a formalism that overcomes those problems. Section 5 presents related work, and Section 6 concludes this paper.

2 WSMO Choreography: An Abstract State Machine Model

WSMO choreography [17] is a state-based model inspired by the methodology of *Abstract State Machine* (ASMs) [3]. It provides basic mechanisms for modeling interactions between service providers and clients at an abstract level. The use of an ASM-based model has several benefits:

- *Minimality*: ASMs are based on a small assortment of modeling primitives.
- *Expressivity*: ASMs can model arbitrary computations.
- *Formality*: ASMs provide a formal framework to express dynamics.

WSMO choreography borrows its basic mechanisms from ASMs. A *signature* defines predicates and functions to be used in the description. *Ground facts* specify the underlying database states. State changes are described using *transition rules*, which specify how the states change by falsifying (deleting) some previously true facts and inserting (making true) some other facts.

In WSMO, signatures are defined using *ontologies*. The ground facts that populate database states are instances of concepts and relations defined by the ontologies. State changes are described in terms of creation of new instances or changes to attribute values of objects.

The transition rules used in WSMO have one of the following forms:

$$\begin{aligned} &- \textbf{if } \mathit{Condition} \textbf{ then } \mathit{Rules} \\ &- \textbf{forall } \mathit{Variables} \textbf{ with } \mathit{Condition} \textbf{ do } \mathit{Rules} \\ &- \textbf{choose } \mathit{Variables} \textbf{ with } \mathit{Condition} \textbf{ do } \mathit{Rules} \end{aligned} \tag{1}$$

The *Condition* part (also called *guard*) is a logical expression, as defined by WSML.[7] The *Rules* part is a set of ASM rules, which can be primitive state changes, like *add*, *delete*, or *update* (modify) a fact. More complex transition rules can be defined with the help of *if-then*, *forall* and *choose* rules. As usual with ASMs, WSMO Choreography rules are evaluated in parallel and the updates are executed in parallel as well. When all rules of an ASM are executed in this way, the ASM makes a *transition* from one database state to another.

A *run* of a WSMO Choreography is a finite or infinite sequence of states, $s_0, s_1, ...$, where s_0 is an initial state of the choreography and, for each $n \geq 0$, the choreography can make a transition from state s_n to s_{n+1}.

In the following section we look at how a WSMO Choreography can be generated from graphical representations of interactions.

3 From Visual Modeling to WSMO Choreography Rules

Although the state-based model of WSMO choreography is appealing for modeling interaction with services, the area of process aware information systems [7] has been dominated by various graphical notations and visual languages for process modeling. Examples of such notations and languages include: UML 2.0 Activity Diagram (AD), Business Process Definition Metamodel (BPDM), Business Process Modeling Notation (BPMN), Business Process Modeling Language (BPML), Event Driven Process Chain (EPC), Petri Nets, etc. Although these tools differ in their expressivity, they share core visual elements for specifying sequential, conditional, and parallel executions. One common way of depicting such core elements is thorough AND/OR graphs, which, in the context of business processes, are referred to as *control flow graphs*.

To illustrate, consider an example from [15] in Figure 1. The figure shows a fairly complex pattern of interaction with a service that sells high ticket items. It includes provisions for optionally giving rebates to customers who fulfill certain requirements as well as a possibility that customers might return the ordered items and receive partial refund. Payment is allowed by credit cards or cheques, and in some cases the service might require those payments to be secured by a credit card (if the available limit exceeds the price) or by providing a guarantor for the payment. Under certain circumstances, the client may get a rebate. The

[7] http://www.wsmo.org/TR/d16/d16.1/v0.3/

figure represents the pattern of interaction with the service using an AND/OR *control flow graph*.

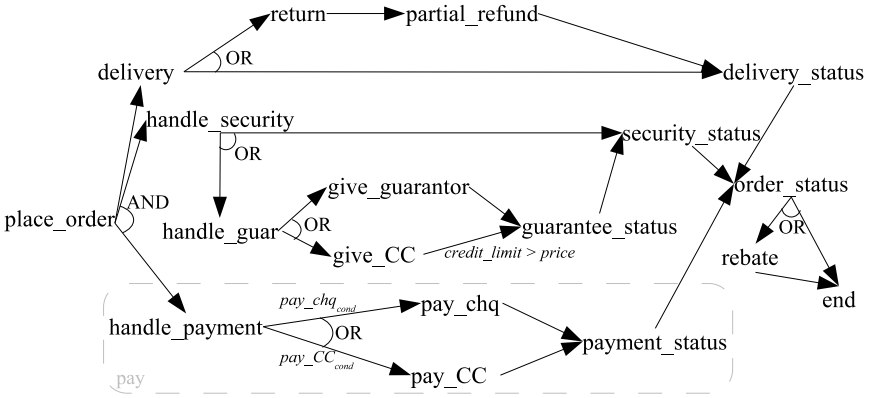

Fig. 1. A process specification.

Control flow graphs are typically used to specify local execution dependencies among the interactions with the service; it is a good way to visualize the overall flow of control. Such a graph has an initial and a final *interaction task*, the successor-interaction for each interaction in the graph (except the final interaction), and a notation that tells whether the successors of an interaction must *all* be executed concurrently (represented by **AND**-split nodes), or whether only one of the alternative branches needs to be executed non-deterministically (represented by **OR**-nodes).

In Figure 1, all successors of the initial interaction **place_order** must be executed, since **place_order** is an **AND**-node. The branches that correspond to these successors eventually join in an **AND**-node called **order_status**. On the other hand, the successors of **place_order** are **OR**-nodes. For example, the successor called **handle_payment** is an **OR**-node, so only one of *its* successors, **pay_chq** *or* **pay_CC**, is supposed to be executed. The node **delivery** is also an **OR**-split. The upper branch hanging from this node represent a situation where a customer accepts delivery but then returns the purchased item. The lower branch, however, has no interactions—it connects **delivery** with the final node on the upper branch, the node **delivery_status**. This means that the upper branch is *optional*: the customer may or may not return the item. Similarly, the segment growing out of the node **order_status** indicates that **rebate** is an optional interaction.

Two more things about the control flow graph in Figure 1 should be noted: the shaded box labeled with **pay**, and the conditions on arcs such as $credit\_limit > price$ attached to the arc leaving the node **give_CC**, and pay_chq_{cond}, pay_CC_{cond} on the arcs adjacent to the node **handle_payment**. Shaded boxes delineate control flow corresponding to *complex interactions* (such as payments) that are

composed of several sub-interactions. If a control graph represents a process then these complex interactions are referred to as complex tasks and correspond to *subprocesses*. The condition $credit\_limit > price$ attached to an arc is called a *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* are 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 details of the representation. Conditions pay_chq_{cond} and pay_chq_{cond} have a similar meaning, but since they are on arcs that start at an **OR**-node, it is assumed that these conditions are mutually exclusive. In general, transition conditions are Boolean expressions attached to the arcs in control flow graphs. Only the arcs whose conditions evaluate to true can be followed at run time. Sometimes a control flow graph may contain additional elements, such as loops, but we do not deal with these here.

In order to represent this example using the ASM-based WSMO Choreography model, we develop a methodology for translating control-flow graphs to WSMO Choreography transition rules. For the purpose of this paper we make the following pragmatic assumptions:

- The signature of a WSMO Choreography consists of:
  - A set of Boolean propositions, where each proposition represents a task in the interaction process;
  - The *cond* predicate which takes two task names as parameters and represents the conditions attached to the arc connecting these two tasks; and
  - The *chosen* predicate which takes a task name as a parameter and is used to model nondeterminism.
- The state of a WSMO Choreography is determined by the truth values of:
  - The propositions corresponding to the tasks in the choreography process. If such a proposition is true, it means that the task has already been executed; if it is false then the task has not yet been executed.
  - The instances of the predicate *cond*. An instance $cond(task_1, task_2)$ evaluates to true if there is an arc from $task_1$ to $task_2$ and the condition attached to that arc evaluates to true. It evaluates to "false" otherwise. If the condition attached to the arc connecting $task_1$ to $task_2$ is not specified explicitly then $cond(task_1, task_2)$ is assumed to be true.
  - The instances of the predicate *chosen*. For each OR-split node, t, with successors $t_1, \ldots, t_n$, it is assumed that exactly one of the predicates $chosen(t_1), \ldots, chosen(t_n)$ evaluates to true. How exactly this evaluation happens is left unspecified. This can be due to an external action or due to the execution of the task t.

Now we can define the translation $\mathfrak{T}$, which transforms a control graph, G, into a set of WSMO Choreography transition rules. The update rules of the form *add(b)* are used to change the truth value of the proposition b to *true*. When b becomes true, it means that the task represented by b has finished its execution.

$$\mathfrak{T}(G) = \begin{cases} \textbf{if } a \wedge \neg b \wedge cond(a,b) \wedge chosen(b) \textbf{ then } add(b), \\ \qquad \text{if there is an arc from } a \text{ to } b \\ \qquad\quad \text{and } a \text{ is an OR-split node} \\ \textbf{if } a_1 \wedge \cdots \wedge a_n \wedge \neg b \wedge cond(a_1,b) \wedge \cdots \wedge cond(a_n,b) \textbf{ then } add(b), \\ \qquad \text{if there is an arc from each } a_1 \ldots a_n \text{ to } b \\ \qquad\quad \text{and } b \text{ is an AND-join node} \\ \textbf{if } a \wedge \neg b \wedge cond(a,b) \textbf{ then } add(b), \\ \qquad \text{if there is an arc from } a \text{ to } b \\ \qquad\quad \text{and } a \text{ is not OR-split node} \\ \qquad\quad \text{and } b \text{ is not AND-join node} \end{cases}$$

Applying this transformation to the example in Figure 1 yields the following. For the arcs originating from **OR**-split nodes we have:

$$\begin{array}{l} \textbf{if } delivery \wedge \neg return \wedge chosen(return) \textbf{ then } add(return) \\ \textbf{if } delivery \wedge \neg delivery\_status \wedge chosen(delivery\_status) \textbf{ then } add(delivery\_status) \\ \textbf{if } handle\_security \wedge \neg security\_status \wedge chosen(security\_status) \\ \quad \textbf{then } add(security\_status) \\ \textbf{if } handle\_security \wedge \neg handle\_guar \wedge chosen(handle\_guar) \textbf{ then } add(handle\_guar) \\ \textbf{if } handle\_guar \wedge \neg give\_guarantor \wedge chosen(give\_guarantor) \\ \quad \textbf{then } add(give\_guarantor) \\ \textbf{if } handle\_guar \wedge \neg give\_CC \wedge chosen(give\_CC) \textbf{ then } add(give\_CC) \\ \textbf{if } handle\_payment \wedge \neg pay\_chq \wedge pay\_chq_{cond} \wedge chosen(pay\_chq) \textbf{ then } add(pay\_chq) \\ \textbf{if } handle\_payment \wedge \neg pay\_CC \wedge pay\_CC_{cond} \wedge chosen(pay\_CC) \textbf{ then } add(pay\_CC) \\ \textbf{if } order\_status \wedge \neg rebate \wedge chosen(rebate) \textbf{ then } add(rebate) \\ \textbf{if } order\_status \wedge \neg end \wedge chosen(end) \textbf{ then } add(end) \end{array} \tag{2}$$

Here we omit those instances of the predicate $cond(t_1, t_2)$, which are not specified explicitly and thus are *true* according to our assumptions. For all the arcs ending in the **AND**-join node, *delivery_status*, we have:

$$\textbf{if } delivery\_status \wedge payment\_status \wedge security\_status \wedge \neg order\_status \textbf{ then } add(order\_status)$$

For the remaining arcs, we have:

$$\begin{array}{l} \textbf{if } place\_order \wedge \neg delivery \textbf{ then } add(delivery) \\ \textbf{if } place\_order \wedge \neg handle\_security \textbf{ then } add(handle\_security) \\ \textbf{if } place\_order \wedge \neg handle\_payment \textbf{ then } add(handle\_payment) \\ \textbf{if } return \wedge \neg partial\_refund \textbf{ then } add(partial\_refund) \\ \textbf{if } partial\_refund \wedge \neg delivery\_status \textbf{ then } add(delivery\_status) \\ \textbf{if } give\_guarantor \wedge \neg guarantee\_status \textbf{ then } add(guarantee\_status) \\ \textbf{if } give\_CC \wedge \neg guarantee\_status \wedge (credit\_limit > price) \\ \quad \textbf{then } add(guarantee\_status) \\ \textbf{if } pay\_chq \wedge \neg payment\_status \textbf{ then } add(payment\_status) \\ \textbf{if } pay\_CC \wedge \neg payment\_status \textbf{ then } add(payment\_status) \\ \textbf{if } rebate \wedge \neg end \textbf{ then } add(end) \end{array}$$

The above WSMO Choreography rules capture the behavior of the process depicted in Figure 1. It is interesting to note that the parallelism represented

in the control flow graph is simulated using concurrent, interleaved execution of tasks.

4 Extending WSMO Choreography with Services Policies and Client Requirements

In the context of service-orientation service providers and requester represent autonomous entities and need to agree on how to interact. Specifications, like the one in Figure 1 provide a mechanism to describe potential interactions between providers and requesters, but service providers and their clients might want to impose additional policies and constraints on those interactions. In this section we explain how such policies can be representations and formulate the problem of agreement between providers and requesters, which is a form of service *contracting*. Then we point out an inadequacy in the current WSMO choreography for representing policies and propose Concurrent Transaction Logic as an extension which allows not only to model service interaction and policies, but also perform certain types of reasoning.

4.1 The Problem of Contracting for Services

In order for service providers and requesters engage in a business deal, they need to agree on the terms of the engagement, or on a *contract*. Since contract requirements can vary and might depend on the client, control-flow graphs are too rigid for representing variable contract terms. These terms often take the form of global temporal and causality constraints between the tasks performed by the services and/or clients. Such constraints coming from the service provider side are called *service policies*, and the constraints coming from the client are called *client contract requirements*. To illustrate, we will again use the choreography depicted in Figure 1. The constraints appropriate for that example are shown in Figure 2.

Service policy:

1. If **pay_CC** (paying by credit card) takes place after accepting **delivery** then **guarantee_status** (giving security) must *precede* **delivery**
2. If **pay_chq** takes place after accepting **delivery** then **pay_chq** (paying by cheque) *immediately* follows **delivery**
3. If **rebate** is given then **pay** must precede accepting **delivery**

Client contract requirement:

4. The interaction of accepting **delivery** must precede **pay_chq**

Fig. 2. Service policies and client contract requirements.

Generally, the terms that may go into a contract can be very complex, but we will focus on temporal and causality constraints, such as the requirement that

certain tasks must all be performed in the course of the execution of a service; that certain tasks must *not* occur together in the same execution; or that some tasks require that certain other tasks execute before or after a given task. We refer the reader to the language of constraints defined in [15] for the specifics of the language used for policy specification.

Once we have a way of describing both the choreographies and constraints, we need a technique for automated support for tasks, such as contracting and enactment, which informally can be defined as follows:

1. **Contracting**: Given a control-flow graph, G, with a set of service policies $C_{service}$ and client requirements C_{client}, the question of determining if a contract between the sides is possible is the problem of finding out if there is an execution of G such that $C_{service}$ and C_{client} are satisfied (i.e. finding out if $G \wedge C_{service} \wedge C_{client}$ is true).
2. **Enactment**: Given a control-flow graph G, with a set of service policies $C_{service}$ and client requirements C_{client}, the problem of enactment is that of generating a sequence of task executions (from G) such that $C_{service}$ and C_{client} are satisfied.

Before we describe a solution to these problems, we need to spend some time looking at the limitations of the current WSMO choreography.

4.2 Limitations of WSMO Choreography

The methodology proposed in Section 3 offers a way of translating control-flow graphs to WSMO choreography rules. To better understand how WSMO will cope with contract requirements, consider the first constraint from Figure 2:

If **pay_CC** (paying by credit card) takes place after accepting **delivery**
then **guarantee_status** (giving security) must *precede* **delivery**

To compile this constraint into WSMO choreography, as defined the rules given in Section 3, the rule that enables *pay_CC* (the 8th rule in (2)) needs to be replaced with the following two rules:

if $handle\_payment \wedge \neg pay\_CC \wedge pay\_CC_{cond} \wedge chosen(pay\_CC) \wedge \neg delivery$ **then** $add(pay\_CC)$
if $handle\_payment \wedge \neg pay\_CC \wedge pay\_CC_{cond} \wedge chosen(pay\_CC) \wedge delivery$
$\wedge\, after(guarantee\_status, delivery)$
then $add(pay\_CC)$

The first rule above ensures that if *pay_CC* was chosen to execute and *delivery* was not executed, the constraint does not apply and thus *pay_CC* can execute. The second rule says that if *pay_CC* has to execute and *delivery* was previously executed, then *pay_CC* may execute only if *delivery* was executed after *guarantee_status* (i.e., if $after(guarantee\_status, delivery)$ is true). To ensure the latter, an additional rule is needed:

$$\textbf{if } place\_order \wedge \neg delivery \wedge guarantee\_status \\ \textbf{then } add(after(guarantee\_status, delivery))$$

Let us now consider the second constraint from Figure 2:

If **pay_chq** takes place after accepting **delivery**
then **pay_chq** (paying by cheque) *immediately* follows **delivery**

To capture this constraint, the WSMO Choreography rule that enables *pay_chq* (the 9th rule in (2)) need to be replaced with two rules as follows:

$$\textbf{if } handle\_payment \wedge \neg pay\_chq \wedge pay\_chq_{cond} \wedge chosen(pay\_chq) \wedge \neg delivery \textbf{ then } add(pay\_chq)$$
$$\textbf{if } handle\_payment \wedge \neg pay\_chq \wedge pay\_chq_{cond} \wedge chosen(pay\_chq) \wedge delivery \wedge last\text{-}task = delivery \\ \textbf{then } add(pay\_chq)$$

The first rule above ensures that if *pay_chq* was chosen to execute and *delivery* was not executed, the constraint does not apply and thus *pay_chq* can execute. The second rule says that if *pay_chq* has to execute and *delivery* was previously executed, then *pay_chq* must execute only if *delivery* was executed immediately prior to that (i.e. *last-task* = *delivery* is true). Now, to ensure that *last-task* always records the last task which executed, all the rules need to be extended such that whenever one executes, it records that it is the last task which executed. That is, whenever we have $add(task)$, we need to add *last-task* = *task*, for example

$$\textbf{if } return \wedge \neg partial\_refund \\ \textbf{then } add(partial\_refund), add(last\text{-}task = partial\_refund) \tag{3}$$

The above exercise should give an idea how much constraints might complicate specification of a WSMO choreography. Even a simple constraint, like the first constraint above, may cause an increase in the number of rules as well as in the number of conditions in the guards of these rules. Adding or deleting constraints will require a recompilation of the entire choreography specification, and such a scenario may quickly become unmanageable. Slightly more complex constraints, like the second constraint above, may lead to further complications, such as the requirement to log the execution of the choreography tasks using auxiliary predicates, such as *last-task*. Additional techniques might need to be developed to handle other constraints described in [15]. All this suggests that constraints must be kept separate from WSMO choreography specifications, if the latter are to be represented as ASMs.

The solution we propose is to use Concurrent Transaction Logic (CTR) [2] as a unifying mechanism for representing both WSMO choreography rules (derived using the methodology in Section 3) and constraints. An added benefit is that CTR supports certain forms of reasoning about such choreographies.

4.3 CTR to the Rescue

Concurrent Transaction Logic (CTR) [2] is a formalism for declarative specification, analysis, and execution of transactional processes. It has been successfully

applied to modeling and reasoning about workflows and services [5, 19, 6, 15]. In this paper we show how CTR can model and extend WSMO choreography, including the ability to specify contract requirements. We first give an informal introduction to CTR and then illustrate its use for the examples in Figures 1 and 2.

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. Apart from the classical connectives $\vee$, $\wedge$, $\neg$, $\forall$, and $\exists$, CTR has two new connectives, $\otimes$ (*serial conjunction*) and $|$ (*concurrent conjunction*), plus one modal operator $\odot$ (*isolated execution*). The intended meaning of the CTR connectives can be summarized as follows:

- $\phi \otimes \psi$ *means*: execute ϕ then execute ψ. In terms of control flow graphs (cf. Figure 1), this connective can be used to formalize arcs connecting adjacent activities.
- $\phi \mid \psi$ *means*: ϕ and ψ must both execute concurrently, in an interleaved fashion. This connective corresponds to the "AND"-nodes in control flow graphs.
- $\phi \wedge \psi$ *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 process specifications.
- $\phi \vee \psi$ *means*: execute ϕ *or* execute ψ non-deterministically. This connective corresponds to the "OR"-nodes in control flow graphs.
- $\neg\phi$ *means*: execute in any way, provided that this will *not* be a valid execution of ϕ. Negation is an important ingredient in the specifications of temporal constraints.
- $\odot\phi$ *means*: execute ϕ in isolation, *i.e.*, without interleaving with other concurrently running activities.

Implication $p \longleftarrow q$ is defined as $p \vee \neg q$. 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 rephrased as: one way to execute p is to execute its definition, q.

The semantics of CTR is based on the idea of *paths*, which are finite sequences of database states. For the purpose of this paper, the reader can think of these states as just relational databases. For instance, if $s_1, s_2, ..., s_n$ are database 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 are assigned 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. Details of the model theory of CTR can be found in [2].

The control flow of a service choreography is represented as a *concurrent-Horn goal*. Subworkflows are defined using *concurrent Horn rules*. We define these notions now.

- A *concurrent Horn goal* is defined recursively as follows:
  - 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.

A service choreography is represented in CTR as a concurrent Horn goal which is built out of atomic formulas that represent interaction tasks or subprocesses. The difference between interaction tasks and subprocesses is that the former are considered atomic primitive tasks, while subprocesses are defined using concurrent-Horn rules. Now we can specify the choreography depicted in Figure 1 as a concurrent Horn goal as follows:

$$\begin{aligned} place\_order \otimes (\,(\,delivery \otimes (refund \vee \texttt{state})\,) \\ \mid (\,security \vee \texttt{state}\,) \\ \mid pay) \;\otimes\; (rebate \vee \texttt{state}) \otimes end \end{aligned}$$

where *security*, *pay*, and $refund$ are subprocesses defined using concurrent concurrent-Horn rules as follows:

$$\begin{aligned} &security \leftarrow \\ &\quad (give\_guarantor \;\vee\; (\,give\_CC \otimes\; credit\_limit \otimes Limit > Price\,)) \\ &\quad \otimes\; guarantee\_status \\ &pay \leftarrow (pay\_chq_{cond} \otimes pay\_chq) \vee (pay\_CC_{cond} \otimes pay\_CC) \\ &refund \leftarrow return \otimes partial\_refund \end{aligned}$$

Here we should point out the use of a special proposition `state`, which is true only on paths of length 1, that is, on database states. It is used in the above to indicate optional actions. Another propositional constant that we will use in the representation of constraints is `path`, defined as $\texttt{state} \vee \neg\texttt{state}$, which is true on every path.

The above representation is much more compact and modular than the corresponding WSMO choreography rules described in Section 3. First, the definitions of *security* and *refund* are significantly simpler and clearer. Second, because of the non-deterministic nature of disjunction in CTR, we can abstract from the actual representations of conditions on the arcs originating from **OR**-nodes and do not need to specify them explicitly. We no longer need nodes such as *delivery_status*, *security_status*, or *order_status*, which were required in the WSMO choreography rules.

CTR can also naturally represent the constraints used for service contracts. For instance, the CTR representation of the constraints from Figure 2 is as follows:

$$\begin{array}{l} 1.\ (\nabla delivery \otimes \nabla pay\_CC) \ \rightarrow \ (\nabla guarantee\_status \otimes \nabla delivery) \\ 2.\ (\nabla delivery \otimes \nabla pay\_chq) \ \rightarrow \ \nabla \odot (delivery \otimes pay\_chq) \\ 3.\ \nabla rebate \rightarrow (\nabla pay \otimes \nabla delivery) \\ 4.\ \nabla delivery \otimes \nabla pay\_chq \end{array} \tag{4}$$

Here $\nabla task$ is a short cut for the formula $\texttt{path} \otimes task \otimes \texttt{path}$, which means that $task$ must happen sometime during the execution of the choreography. The formula $\texttt{path} \otimes \odot(task_1 \otimes \cdots \otimes task_n) \otimes \texttt{path}$ means that tasks $task_1, ..., task_n$ must execute next to each other with no other events in-between. Observe that although representing such constraints complicates WSMO choreography rules even further (and is not modular), their representation in CTR is simple and natural.

Now, to specify that a choreography specification must obey the terms of a contract, we must simply build a conjunction of the choreography (which is specified as a concurrent Horn goal) and the constraints that form the service contract specification, i.e., $G \wedge C_{service} \wedge C_{client}$, where G is a choreography specification, $C_{service}$ are the service policies, and C_{client} are the client contract requirements.

A technique for proving formulas of the form $ConcurrentHornGoal \wedge Constraints$ and to constructively find a sequence of states on which $ConcurrentHornGoal \wedge Constraints$ is true (i.e., enactment of the choreography subject to the contract requirements) was initially proposed in [5] and further developed in [15].

In this section we pointed put the limitations of the current ASMs-based WSMO choreography model and showed a way to overcome these problems with the help of CTR. Another advantage of CTR over the ASMs-based model is that any ASM can be mapped into a CTR formula and thus using CTR instead of ASMs does not limit the generality of WSMO choreography. This line of research is pursued in [16].

5 Related Work

To the best of our knowledge, this paper is the first to introduce a methodology that enables reasoning about WSMO choreography, including automatic contracting and enactment of WSMO services.

Automatic service contracting and enactment have been identified as core tasks in the context of semantic Web services (see e.g. [10]), however no approaches have been proposed that directly deal with such tasks. Several reasoning and verification techniques have been developed for the OWL-S [20] Process Model—the counterpart of WSMO choreography in OWL-S. For example, [13] proposed a Petri Net-based operational semantics for the OWL-S Process Model, and applied existing Petri Net analysis techniques to verify process models for

various properties such as reachability, deadlocks, etc. In [1], a verification technique of OWL-S Process Models using model checking is proposed. A planning technique for automated composition of Web services, described in OWL-S process models, is presented in [21]. Although these works are related, they are complementary to the task of service contracting. The Semantic Web Services Framework [18] introduces a rich behavioral process model based on the Process Specification Language PSL [11]. Although PSL is defined in first order logic, which in theory makes behavioral specifications in PSL amenable to automated reasoning, we are not aware of specific works that deal with service contracting in PSL.

In the broader area of Web services, reasoning and verification about service choreographies (especially in the context of WS-CDL[8]) have been investigated in various works, such as [4] and [14]. As in the case of previous works on the OWL-S Process Model, such approaches deal with verification and analysis of various properities of service interactions, but leave the problem of service contracting out. Compared to such approaches, the work presented in this paper deals with a specific choreography model developed in the context of WSMO and proposes a concrete mechanism for service contracting using this model.

Web service policies is another related area. WS-Policy [8] provides a general purpose model to describe the policies of entities in a Web services-based system. Although WS-Policy is widely adopted in industry, it is primarily used as a syntactic mechanism for policy specification. Instead, our approach works at a semantic level and, although focused on a specific choreography model, it enables automatic service contracting and enactment. Other Semantic Web-based policy frameworks, such as KAoS [22] or Rein [12], focus on authentication and authorization policies, whereas our approach targets service choreographies.

Last but not least, our contribution can be considered in the context of the emerging area of compliance checking between business processes and business contracts [9].

6 Conclusions

Semantic Web services (SWS) promise to bring high degree of automation to service discovery, contracting, enactment, monitoring, and mediation. WSMO is a major initiative in this area, which is supported by a number of academic institutions and companies. As with other SWS approaches (e.g. OWL-S, SWSF), modeling and reasoning about service behavior are central to the WSMO framework. While other parts of WSMO are based on a logic, WSMO choreography relies on Abstract State Machines (ASMs) for its formalization. ASMs have many advantages, but they are hard to reason about and there are no methodologies for building ASM-based process specifications that choreograph the interactions with Web services. To overcome these problems, this paper developed a methodology, based on control-flow graphs, for designing WSMO choreographies. We highlighted the limitations of the current WSMO choreography model and

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

showed a way to remedy these problems with the help of Concurrent Transaction Logic. In this way, we enabled reasoning about service contracts and enactment in WSMO and opened up several new opportunities for future work. For example, integration of WSMO Web service discovery[9] with service contracting and enactment is one such opportunity. Verification and optimization of choreography interfaces is another promising direction.

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 by US Army Research Office under a subcontract from BNL. Dumitru Roman was partly funded by the BIT Institute, the SUPER project (FP6-026850), and the Knowledge Web project (FP6-507482).

References

1. A. Ankolekar, M. Paolucci, and K. P. Sycara. Towards a formal verification of owl-s process models. In *International Semantic Web Conference*, pages 37–51, 2005.
2. A. J. Bonner and M. Kifer. Concurrency and Communication in Transaction Logic. In *Joint International Conference and Symposium on Logic Programming*, 1996.
3. E. Börger and R. F. Stärk. *Abstract State Machines—A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.
4. M. Carbone, K. Honda, and N. Yoshida. Structured global programming for communication behaviour. In *16th European Symposium on Programming (ESOP'07)*, 2007.
5. H. Davulcu, M. Kifer, C. R. Ramakrishnan, and I. V. Ramakrishnan. Logic Based Modeling and Analysis of Workflows. In *PODS*, pages 25–33, 1998.
6. H. Davulcu, M. Kifer, and I.V. Ramakrishnan. CTR–S: A Logic for Specifying Contracts in Semantic Web Services. In *WWW2004*, pages 144+, 2004.
7. M. Dumas, W. M. van der Aalst, and A. H. ter Hofstede (eds.). *Process-aware information systems: bridging people and software through process technology*. John Wiley & Sons, Inc., 2005.
8. A. Vedamuthu (editor) et al. Web Services Policy 1.5 – Framework (WS-Policy). W3c recommendation, W3C, September 2007. Available at http://www.w3.org/TR/ws-policy.
9. G. Governatori, Z. Milosevic, and S. Sadiq. Compliance checking between business processes and business contracts. In *EDOC '06*, pages 221–232. IEEE Computer Society, 2006.
10. B. Grosof, M. Gruninger, M. Kifer, D. Martin, D. McGuinness, B. Parsia, T. Payne, and A. Tate. Semantic Web Services Language Requirements Version 1. Working draft, SWSI Language Committee, 2005. Available at http://www.daml.org/services/swsl//requirements/swsl-requirements.shtml.
11. M. Grüninger and C. Menzel. The process specification language (PSL) theory and applications. *AI Mag.*, 24(3):63–74, 2003.
12. L. Kagal, T. Berners-Lee, D. Connolly, and D. J. Weitzner. Using Semantic Web Technologies for Policy Management on the Web. In *AAAI*, 2006.

[9] http://www.wsmo.org/TR/d5/d5.1/

13. S. Narayanan and S. McIlraith. Simulation, verification and automated composition of web services. In *Proceedings of the 11th International World Wide Web Conference (WWW2002)*, Honolulu, Hawaii, May 2002.
14. Z. Qiu, X. Zhao, C. Cai, and H. Yang. Towards the theoretical foundation of choreography. In *WWW*, pages 973–982, 2007.
15. D. Roman and M. Kifer. Reasoning about the Behavior of Semantic Web Services with Concurrent Transaction Logic. In *VLDB*, 2007.
16. D. Roman and M. Kifer. Simulating Abstract State Machines (ASMs) with Concurrent Transaction Logic (CTR). WSMO Deliverable D14.2v0.1. Technical report, DERI Innsbruck, 2007. Available at http://www.wsmo.org/TR/d14/d14.2/v0.1/d14.2v01.pdf.
17. D. Roman, J. Scicluna, and J. Nitzsche (Eds.). Ontology-based Choreography. WSMO Deliverable D14v0.1. Technical report, DERI Innsbruck, 2007. Available at http://www.wsmo.org/TR/d14/v1.0/.
18. Semantic Web Services Framework. SWSF Version 1.0. Available from http://www.daml.org/services/swsf/1.0/, 2005.
19. P. Senkul, M. Kifer, and I. Toroslu. A Logical Framework for Scheduling Workflows under Resource Allocation Constraints. In *VLDB 2002*, pages 694–705, 2002.
20. The OWL Services Coalition. OWL-S 1.1 beta release. Available from http://www.daml.org/services/owl-s/1.1B/, July 2004.
21. P. Traverso and M. Pistore. Automated composition of semantic web services into executable processes. In *International Semantic Web Conference*, pages 380–394, 2004.
22. A. Uszok, J. M. Bradshaw, R. Jeffers, A. Tate, and J. Dalton. Applying KAoS Services to Ensure Policy Compliance for Semantic Web Services Workflow Composition and Enactment. In *International Semantic Web Conference*, 2004.