$\mathcal{CTR}$-S: A Logic for Specifying Contracts in Semantic Web Services

Hasan Davulcu
Department of CSE
Arizona State University
Box 875406,
Tempe, AZ 85287-5406
hdavulcu@asu.edu

Michael Kifer
Department of Computer Science
Stony Brook University
Stony Brook, NY 11794
kifer@cs.stonybrook.edu

I.V. Ramakrishnan
Department of Computer Science
Stony Brook University
Stony Brook, NY 11794
ram@cs.stonybrook.edu

ABSTRACT

A requirements analysis in the emerging field of *Semantic Web Services* (SWS) (see http://daml.org/services/swsl/requirements/) has identified four major areas of research: intelligent *service discovery*, automated *contracting of services*, *process modeling*, and *service enactment*. This paper deals with the intersection of two of these areas: process modeling as it pertains to automated contracting. Specifically, we propose a logic, called $\mathcal{CTR}$-S, which captures the *dynamic aspects* of contracting for services. Since $\mathcal{CTR}$-S is an extension of the classical first-order logic, it is well-suited to model the static aspects of contracting as well. A distinctive feature of contracting is that it involves two or more parties in a potentially adversarial situation. $\mathcal{CTR}$-S is designed to model this adversarial situation through its novel model theory, which incorporates certain game-theoretic concepts. In addition to the model theory, we develop a proof theory for $\mathcal{CTR}$-S and demonstrate the use of the logic for modeling and reasoning about Web service contracts.

Categories and Subject Descriptors

H.4.m [**Information Systems**]: Miscellaneous; I.1.3 [**Computing Methodologies**]: Symbolic and algebraic manipulation—*Languages and Systems*

General Terms

Web Services, Semantic Web

Keywords

Web Services, Services Composition, Contracts

1. INTRODUCTION

A Web service is a process that interacts with the client and other services to achieve a certain goal. A requirements analysis in the emerging field of *Semantic Web Services* (SWS)[1] has identified three major areas of research: intelligent *service discovery*, automated *contracting of services*, *process modeling*, and *service enactment*. It is generally agreed that Semantic Web Services should be based on a formalism with a well-defined model-theoretic semantics, *i.e.*, on some sort of a logic. In this paper we propose a logic, called $\mathcal{CTR}$-S, which captures the *dynamics* of contracting for services and thus is in the intersection of the areas of contracting and process modeling. Since $\mathcal{CTR}$-S is an extension of the classical first-order logic, it is well-suited for modeling of the static aspects of contracting as well. If object-oriented representation is desired, F-logic [14] (and an adaptation of $\mathcal{CTR}$-S) can be used instead.

The idea of using a logic to model processes is not new [8, 2, 12, 23, 21]. These methodologies are commonly based on the classical first-order logic, temporal logic [9], and Concurrent Transaction Logic [5]. A distinctive aspect of contracting in Web services, which is not captured by these formalisms, is that contracting involves multi-party processes, which can be adversarial in nature. One approach to deal with this situation could be to try and extend a multi-modal logic of knowledge [10]. However, we found it more expedient to extend Concurrent Transaction Logic [5] (or $\mathcal{CTR}$), which has been proven a valuable tool for modeling and reasoning about processes [8, 4, 17]. The extension is called $\mathcal{CTR}$-S and is designed to model the adversarial situation that arises in service contracting. This is achieved by extending the model theory of $\mathcal{CTR}$ with certain concepts borrowed from the Game Theory [18, 13, 19]. In this paper we also develop a proof theory for $\mathcal{CTR}$-S and illustrate the use of this logic for modeling and reasoning about Web service contracts.

A typical situation in contracting where different parties may sometimes have conflicting goals is when a buyer interacts with a seller and a delivery service. The buyer needs to be assured that the goods will either be delivered (using a delivery service) or money will be returned. The seller might need assurance that if the buyer breaks the contract then part of the down-payment can be kept as compensation. We thus see that services can be adversarial to an extent. Reasoning about such services is an unexplored research area and is the topic of this paper.

Overview and summary of results. We introduce the game theoretic aspects into $\mathcal{CTR}$ using a new connective, the *opponent's conjunction*. This connective represents the choice of action that can be made by a party other than the reasoner. The reasoner here can be the client of a Web service who needs to verify that her goals are met or a service that needs to make sure that its business rules are satisfied no matter what the other parties do. We then develop a model theory for $\mathcal{CTR}$-S and show how this new logic can be used to specify executions of services that may be non-cooperating and have potentially conflicting goals. We also discuss reasoning about a fairly large class of temporal and causality constraints.

In $\mathcal{CTR}$-S, a contract is modeled as a *workflow* that represents the various possibilities for the reasoner and the outside actors. The

[1] See http://daml.org/services/swsl/requirements/

WWW2004, May 17–22, 2004, New York, NY USA.
ACM xxx.xxx.

$\mathcal{CTR}$-S model theory characterizes all possible *outcomes* of a formula $\mathcal{W}$ that represents such a workflow. A constraint Φ characterizes executions of the contract with certain desirable properties (for instance, that either the good is delivered or payment refunded). The formula $\mathcal{W} \wedge \Phi$ characterizes those executions that satisfy the constraint Φ. If $\mathcal{W} \wedge \Phi$ is satisfiable, *i.e.*, there is at least one execution O in its model, then we say that the constraint Φ is *enforcible* in workflow formula $\mathcal{W}$.

We describe a synthesis algorithm that converts declarative specifications, such as $\mathcal{W} \wedge \Phi$, into equivalent $\mathcal{CTR}$-S formulas that can be executed more efficiently and without backtracking. The transformation also detects unsatisfiable specifications, which are contracts that the reasoner cannot force to execute with desirable outcomes. In game-theoretic terms, the result of such a transformation can be thought of as a concise representation of all *winning strategies, i.e.*, all the ways for the reasoner to achieve the desired outcome, regardless of what the rest of the system does, if all the parties obey the terms of the contract.

Finally, since $\mathcal{CTR}$-S is a natural generalization of $\mathcal{CTR}$, a pleasing aspect of this work is that our earlier results in [8] become special cases of the new results.

The rest of the paper is organized as follows. In Section 2, we introduce $\mathcal{CTR}$-S and discuss its use for modeling workflows and contracts. In Section 3, we introduce the model theory of $\mathcal{CTR}$-S. The proof theory of $\mathcal{CTR}$-S is not discussed here due to lack of space, but it is available in the following report.[2] Section 4 introduces causal and temporal workflow constraints that can be used to specify contracts. In Section 5, we present the coordinator synthesis algorithm and discuss its complexity. Section 6 concludes with a discussion of related formalisms.

2. $\mathcal{CTR}$-S AND THE DYNAMICS OF SERVICE CONTRACTS

Familiarity with $\mathcal{CTR}$ [5] can help understanding of $\mathcal{CTR}$-S and its relationship to workflows and contracts. However, this paper is self-contained and includes all the necessary definitions. We first describe the syntax of $\mathcal{CTR}$-S and then supply intuition to help the reader make sense out of the formal definitions. The model theory of $\mathcal{CTR}$-S, which is a rather radical extension of that of $\mathcal{CTR}$, is described in Section 3.

2.1 Syntax

The ***atomic formulas*** of $\mathcal{CTR}$-S are the same as in classical logic, i.e. they are expressions of the form $p(t_1, ..., t_n)$, where *p* is a predicate symbol and t_i are terms. Complex formulas are built with the help of connectives and quantifiers: if ϕ and ψ are $\mathcal{CTR}$-S formulas, then so are $\phi \wedge \psi$, $\phi \otimes \psi$, $\phi \mid \psi$, $\neg\phi$, $\phi \sqcap \psi$, and $(\forall X)\phi$, where X is a variable. Intuitively, the formula $\phi \otimes \psi$ means: execute ϕ and then execute ψ. The connective $\otimes$ is called ***serial conjunction***. The formula $\phi \mid \psi$ denotes an interleaved execution of two games ϕ and ψ. The connective $\mid$ is called ***concurrent conjunction***. The formula $\phi \sqcap \psi$ means that the opponent chooses whether to execute ϕ or ψ, and therefore $\sqcap$ is called ***opponent's conjunction***. The meaning of $\phi \vee \psi$ is similar, except the reasoner makes the decision. In $\mathcal{CTR}$ this connective is called ***classical disjunction*** but because of its interpretation as reasoner's choice we will also refer to it as ***reasoner's disjunction***. Finally, the formula $\phi \wedge \psi$ denotes execution of ϕ *constrained* by ψ (or vice versa). It is called ***classical conjunction***.[3]

As in classical logic, we introduce convenient abbreviations for complex formulas: $\phi \leftarrow \psi$ stands for $\phi \vee \neg\psi$ and is called a ***rule***; $\phi \vee \psi$ denotes $\neg(\neg\phi \wedge \neg\psi)$; and $\exists\, \phi$ is an abbreviation for $\neg\forall\neg\phi$. The opponent's conjunction also has a dual connective, $\sqcup$, but we will not discuss its properties in this paper.

As mentioned in the introduction, we model the dynamics of service contracts using the abstraction of a 2-party workflow, where the first party is the reasoner and the other represents the rest of the players involved in the contract. In general, if several parties need to be able to reason about the same contract, the contract can be represented as several 2-party workflows, each representing the contract from the point of view of a different reasoner.

Definition 1. (**Workflows**) A ***$\mathcal{CTR}$-S goal*** is either an atomic formula or an expression of the form $\phi \otimes \psi$, $\phi \mid \psi$, $\phi \vee \psi$, or $\phi \sqcap \psi$, where ϕ and ψ are $\mathcal{CTR}$-S goals. A ***rule*** is of the form $head \leftarrow body$, where *head* is an atomic formula and *body* a $\mathcal{CTR}$-S goal. A ***workflow control specification*** consists of a $\mathcal{CTR}$-S $goal$ and a (possibly empty) set of $rules$.

In this paper we will restrict workflows to be non-recursive, so having rules is just a matter of convenience, which does not affect the expressive power. Note also that some connectives, like $\wedge$, are not allowed in workflow control specifications, but are used to specify constraints.

2.2 Modeling Contract Dynamics in $\mathcal{CTR}$-S

Example 1. (**Procurement Contract**) Consider a procurement application that consists of a $buyer$ interacting with three services, $seller$, $financing$ and $delivery$. We assume that the $buyer$ is the reasoner for the rest of this example.

Services are modeled in terms of their *significant events*. Thus, the $buyer$ service begins when the significant event pay_escrow occurs. Then concurrent execution of $seller$, and $financing$ services proceeds.

Thus, at a high level, this $buyer$ workflow can be represented as:

$$pay\_escrow \otimes (financing \mid seller)$$

The connective $\otimes$ represents succession of events or actions: when the above expression "executes," the underlying database state changes first to one where the workflow has started (which would typically be expressed by insertion of a literal pay_escrow into the log and execution of an escrow payment transaction), then to one where the execution of the rest of the formula proceeds. The connective $\mid$ represents concurrent, *interleaved* execution of the two sequences of actions. Intuitively, this means that a legal execution of $(financing \mid seller)$ is a sequence of database states where the initial subsequence corresponds, say, to a partial execution of the subformula $financing$; the next subsequence of states corresponds to the execution of the $seller$; the following subsequence is a continuation of the execution of the $financing$; and so on. The overall execution sequence of $(financing \mid seller)$ is a merge of an execution sequence for the left subformula and an execution sequence for the right subformula.

Execution has precise meaning in the model and proof theory of $\mathcal{CTR}$. Truth of $\mathcal{CTR}$ formulas is established *not* over database states, as in classical logic, but over sequences of states. Truth of a formula over such a sequence is interpreted as an execution of that formula in which the initial database state of the sequence is successively changed to the second, third, etc., state. The database

[2] http://www.public.asu.edu/∼hdavulcu/gamectrtech.pdf

[3] The aforesaid meaning of $\wedge$ is all but classical. However, its semantic definition looks very much like that of a conjunction in predicate calculus. This similarity is the main reason for the name.

ends up in the final state of the sequence when the execution terminates.[4]

Workflow formulas can be modularized with the help of the rules. The intuitive meaning of a rule, $head \leftarrow body$, where *head* is an atomic formula and *body* is a $\mathcal{CTR}$-S goal, is that *head* is an invocation interface to *body*, which is viewed as a subroutine. This is because according to the semantics described in Section 3, such a rule is true if every legal execution of *body* must also be a legal execution of *head*. Combined with the minimal model semantics this gives the desired effect [6]. With this in mind, we can now express the above procurement workflow as follows:

$$buyer \longleftarrow pay\_escrow \otimes (seller \mid financing)$$

Next, $buyer$ searches for matching services for $seller$ using a services directory to discover the following match.

$$seller \longleftarrow reserve\_item \otimes (delivery \vee collect\_escrow)$$
$$delivery \longleftarrow insured \vee uninsured$$

The definition of $seller$ involves the use of the $\vee$ connective. The connective $\vee$ here represents alternative executions for the reasoner. For instance, a legal execution of $insured \vee uninsured$ is either a legal execution of $insured$ or of $uninsured$. Similarly, a legal execution of $seller$ involves the execution of $reserve\_item$ and then, the alternative execution of either $delivery$ or $recv\_escrow$.

The above definition of $seller$ also requires compliance to the following contract requirements between the $buyer$ and the $seller$:

- if *buyer* cancels, then *seller* keeps the escrow
- if *buyer* pays, then *seller* must deliver

Thus, $\vee$ connective represents a *choice*. The question is, however, whose choice it is: the reasoner's or that of the rest of the actors? In an environment where workflow activities might not always cooperate, we need a way to distinguish these two kinds of choices. For instance, the contract may say that the outcome of the actions of the $delivery$ agent the goods might be $delivered$ or $lost$. This alternative is clearly not under the control of the reasoner. On the other hand, the choice of whether to use an $insured$ or of $uninsured$ delivery is made by the reasoner. With this understanding, the $insured$ and $uninsured$ services are defined as follows:

$$insured \longleftarrow (delivered \otimes confirm) \sqcap (lost \otimes confirm)$$
$$uninsured \longleftarrow (delivered \otimes confirm) \sqcap lost$$

The above connective $\sqcap$ represents the service environment's choice. If the $buyer$ uses $insured\ delivery$ then he is guaranteed a confirmation whether the item is $delivered$ or $lost$. If the $buyer$ uses $uninsured$ delivery then he can get a confirmation only if the item is $delivered$. Whether the item is $delivered$ or $lost$ depends on the service environment's choice.

Next, the $buyer$ identifies the following matching service for $financing$:

$$financing \longleftarrow (approve \otimes (make\_payment \vee cancel)) \sqcap (reject \otimes cancel)$$

Note that the approval or rejection of the financing request is the servicing agent's choice. However, if the financing is approved the decision of whether to proceed and $make\_payment$ or to $cancel$ the procurement still depends on the choice of the reasoner, the $buyer$ agent. In addition, the $financing$ agent might require the following clause in the contract:

- if *financing* is approved and *buyer* does not $cancel$ then *delivery* should *confirm*

The details of how to express the above contract in $\mathcal{CTR}$-S will be explained in Section 4. Also in Section 5 we will show that regardless of the service environment's behavior, a suitable workflow coordinator for *enforcing* the above contract can be synthesized automatically.

We shall see that a large class of temporal and causality constraints (including those in our example) can be represented as $\mathcal{CTR}$-S formulas. If Φ represents such a formula for the above example, then finding a strategy to win the above game (*i.e.*, enforce the constraints) is tantamount to checking whether $buyer \wedge \Phi$ is satisfiable in $\mathcal{CTR}$-S.

Before going on, we should clear up one possible doubt: why is there only one opponent? The answer is that this is sufficient for a vast majority of practical cases, especially those that arise in Web services. Indeed, even when multiple independent actors are involved, we can view each one of them (or any group that decides to cooperate) as the *reasoner* and all the rest as the *opponent*. Any such actor or a group can then use $\mathcal{CTR}$-S to verify that its goals (specified as a condition Φ) are indeed enforcible.

3. MODEL THEORY

In this section we define a model theory for our logic. The importance of a model theory is that it provides the exact semantics for the behavioral aspects of service contracts and thus serves as a yardstick of correctness for the algorithms in Section 5.

3.1 Sets of Multipaths

A $\boldsymbol{path}$ is a sequence of database states, $d_1 ... d_n$. Informally, a $\boldsymbol{database\ state}$ can be a collection of facts or even more complex formulas, but for this paper we can think of them as simply symbolic identifiers for various collections of facts.

In $\mathcal{CTR}$ [5], which allows concurrent, interleaved execution, the semantics is based on *sequences* of paths, $\pi = \langle p_1, ..., p_m \rangle$, where each p_i is a path. Such a sequence is called a $\boldsymbol{multipath}$, or an $\boldsymbol{m\text{-}path}$ [5]. For example, $\langle d_1 d_2,\ d_3 d_4 d_5 \rangle$ is an m-path that consists of two paths: one having two database states and the other three (note that a comma separates paths, not states in a path). As explained in Example 1, multipaths capture the idea of an execution of a transaction that interleaves with executions of other transactions. Thus, an m-path can be viewed as an execution that is broken into segments, such that other transactions could execute in-between the segments.

$\mathcal{CTR}$-S further extends this idea by recognizing that in the presence of other parties, the reasoner cannot be certain which execution (or "play") will actually take place, due to the lack of information actual opponent's moves. However, the reasoner can have a *strategy* to ensure that regardless of what the opponent does the resulting execution will be contained within a certain set of plays. Such a set is called an $\boldsymbol{outcome\ of\ the\ game}$. Thus, while truth values of formulas in Transaction Logic are determined on paths and of $\mathcal{CTR}$ formulas on m-paths, $\mathcal{CTR}$-S formulas get their truth values on $\boldsymbol{sets\ of\ m\text{-}paths}$. Each such set, S, is interpreted as an outcome of the game in the above sense, and saying that a $\mathcal{CTR}$-S formula, ϕ, is true on S is tantamount to saying that S is an outcome of ϕ. In particular, two games are equivalent if and only if they have the same sets of outcomes.

[4] Space limitation does not permit us to compare $\mathcal{CTR}$ to other logics that on the surface might appear to address the same issues (*e.g.*, temporal logics, process and dynamic logics, the situation calculus, etc.). We refer the reader to the extensive comparisons provided in [5, 6].

3.2 Satisfaction on Sets of Multipaths

The following definitions make the above discussion precise.

Definition 2. (**m-Path Structures**) An $\boldsymbol{m\text{-}path\ structure}$ [5] is a triple of the form $\langle U, I_{\mathcal{F}}, I_{path}\rangle$, where U is the domain, $I_{\mathcal{F}}$ is an interpretation function for constants and function symbols (exactly like in classical logic), and I_{path} is a mapping such that if π is an m-path, then $I_{path}(\pi)$ is a first-order semantic structure (as commonly used in classical predicate logic).

For a $\mathcal{CTR}$ formula, ϕ, and an m-path, π, the truth of ϕ on π with respect to an m-path structure is determined by the truth values of the components of ϕ on the appropriate sub-m-paths π_i on π. In a well-defined sense, establishing the truth of a formula, ϕ, over an m-path, $\pi = \langle p_1, ..., p_n\rangle$, corresponds to the possibility of executing ϕ along π where the gaps between p_1, ..., p_n are filled with the executions of other formulas [5].

The present paper extends this notion to $\mathcal{CTR}$-S by defining truth of a formula ϕ over *sets* of m-paths, where each such set represents a possible outcome of the game represented by ϕ. The new definition reduces to $\mathcal{CTR}$'s for formulas that have no $\sqcap$'s.

Definition 3. (**Satisfaction**) Let $\mathbf{I} = \langle U, I_{\mathcal{F}}, I_{path}\rangle$ be an m-path structure, π be an arbitrary m-path, S, T, S_1, S_2, etc., be *non-empty* sets of m-path, and let ν be a variable assignment (*i.e.*, it assigns an element of U to each variable). We define the notion of $\boldsymbol{satisfaction}$ of a formula, ϕ, in $\mathbf{I}$ on S by structural induction on ϕ:

1. **Base Case:** $\mathbf{I}, \{\pi\} \models_\nu p(t_1, \ldots, t_n)$ if and only if $I_{path}(\pi) \models_\nu^{classic} p(t_1, \ldots, t_n)$, for any atomic formula $p(t_1, \ldots, t_n)$, where $\models^{classic}$ is the usual first-order entailment. (Recall that $I_{path}(\pi)$ is a usual first-order semantic structure, so $\models^{classic}$ is defined for it.)

   Typically, $p(t_1, ..., t_n)$ is either defined via rules (as in Example 1) or is a "built-in," such as $insert(q(a,b))$, with a fixed meaning. For instance, in case of $insert(q(a,b))$ the meaning would be that $\mathbf{I}, \{\pi\} \models_\nu insert(q(a,b))$ iff π is an m-path of the form $\langle d_1\, d_2\rangle$ (*i.e.*, it consists of a single path), where $d_2 = d_1 \cup \{q(a,b)\}$.[5] These built-ins are called $\boldsymbol{elementary\ updates}$ and constitute the basic building blocks from which more complex actions, such as those at the end of Example 1, are constructed.

2. **Negation:** $\mathbf{I}, S \models_\nu \neg\phi$ if and only if it is *not* the case that $\mathbf{I}, S \models_\nu \phi$.

3. **Reasoner's Disjunction:** $\mathbf{I}, S \models_\nu \phi \vee \psi$ if and only if $\mathbf{I}, S \models_\nu \phi$ or $\mathbf{I}, S \models_\nu \psi$.
   The dual connective, $\phi \wedge \psi$, is a shorthand for, $\neg(\neg\phi \vee \neg\psi)$.

4. **Opponent's Conjunction:** $\mathbf{I}, S \models_\nu \phi \sqcap \psi$ if and only if $S = S_1 \cup S_2$, for some pair of m-path sets, such that $\mathbf{I}, S_1 \models_\nu \phi$, and $\mathbf{I}, S_2 \models_\nu \psi$. The dual connective, $\sqcup$, also exists, but is not used in this paper.

5. **Serial Conjunction:** $\mathbf{I}, S \models_\nu \phi \otimes \psi$ if and only if there is a set R of m-paths, such that S can be represented as $\bigcup_{\pi\in R} \pi \circ T_\pi$, where each T_π is a set of m-paths, $\mathbf{I}, R \models_\nu \phi$, and for each T_π, $\mathbf{I}, T_\pi \models_\nu \psi$.
   Here $\pi \circ T = \{\pi \circ \delta \mid \delta \in T\}$, where $\pi \circ \delta$ is an m-path obtained by appending the m-path δ to the end of the m-path π. (For instance, if $\pi = \langle d_1 d_2,\ d_3 d_4\rangle$ and $\delta = \langle d_5 d_6,\ d_7 d_8 d_9\rangle$ then $\pi \circ \delta = \langle d_1 d_2,\ d_3 d_4,\ d_5 d_6,\ d_7 d_8 d_9\rangle$.) In other words, R is a set of prefixes of the m-paths in S.

6. **Concurrent Conjunction:** $\mathbf{I}, S \models_\nu \phi \mid \psi$ if and only if there is a set R of m-paths, such that S can be represented as $\bigcup_{\pi\in R} \pi \| T_\pi$, where each T_π is a set of m-paths, and either
   - $\mathbf{I}, R \models_\nu \phi$ and for all T_π, $\mathbf{I}, T_\pi \models_\nu \psi$; or
   - $\mathbf{I}, R \models_\nu \psi$ and for all T_π, $\mathbf{I}, T_\pi \models_\nu \phi$

   Here $\pi \| T_\pi$ denotes the set of *all* m-paths that are interleavings of π with an m-path in T_π. For instance, if $\pi = \langle d_1 d_2,\ d_3 d_4\rangle$ and $\langle d_5 d_6,\ d_7 d_8 d_9\rangle \in T_\pi$ then *one of* the interleavings is $\langle d_1 d_2,\ d_5 d_6,\ d_3 d_4,\ d_7 d_8 d_9\rangle$.

7. **Universal Quantification:** $\mathbf{I}, \pi \models_\nu \forall X.\phi$ if and only if $\mathbf{I}, \pi \models_\mu \phi$ for *every* variable assignment μ that agrees with ν everywhere except on X. Existential quantification, $\exists X.\phi$, is a shorthand for $\neg\forall X \neg\phi$.

Example 2. (**Database Transactions**) Consider the following formula

$$\phi = insert(st) \otimes (insert(ab) \sqcap insert(cm)) \otimes (insert(cp) \vee insert(no))$$

Then the possible outcomes for ϕ can be computed from the outcomes of its components as follows:

1. By (1) in Definition 3: $\{\langle d\ \ d \cup \{st\}\rangle\} \models insert(st)$, $\{\langle d\ \ d \cup \{ab\}\rangle\} \models insert(ab)$, $\{\langle d\ \ d \cup \{cm\}\rangle\} \models insert(cm)$, $\{\langle d\ \ d \cup \{cp\}\rangle\} \models insert(cp)$, and $\{\langle d\ \ d \cup \{no\}\rangle\} \models insert(no)$

2. By (3) in Definition 3: $\{\langle d\ \ d \cup \{cp\}\rangle\} \models (insert(cp) \vee insert(no))$, and $\{\langle d\ \ d \cup \{no\}\rangle\} \models (insert(cp) \vee insert(no))$

3. By (5) in Definition 3: $\{\langle d\ \ d \cup \{ab\}\rangle,\ \langle d\ \ d \cup \{cm\}\rangle\} \models (insert(ab) \sqcap insert(cm))$

4. By (6) in Definition 3: $\{\langle \emptyset\ \{st\}\ \{st, ab\}\rangle,\ \langle \emptyset\ \{st\}\ \{st, cm\}\rangle\} \models insert(st) \otimes (insert(ab) \sqcap insert(cm))$

5. By (6) in Definition 3:
   $\{\langle \emptyset\ \{st\}\ \{st, ab\}\ \{st, ab, cp\}\rangle,\ \langle \emptyset\ \{st\}\ \{st, cm\}\ \{st, cm, cp\}\rangle\} \models \phi$, and
   $\{\langle \emptyset\ \{st\}\ \{st, ab\}\ \{st, ab, cp\}\rangle,\ \langle \emptyset\ \{st\}\ \{st, cm\}\ \{st, cm, no\}\rangle\} \models \phi$, and
   $\{\langle \emptyset\ \{st\}\ \{st, ab\}\ \{st, ab, no\}\rangle,\ \langle \emptyset\ \{st\}\ \{st, cm\}\ \{st, cm, cp\}\rangle\} \models \phi$, and
   $\{\langle \emptyset\ \{st\}\ \{st, ab\}\ \{st, ab, no\}\rangle,\ \langle \emptyset\ \{st\}\ \{st, cm\}\ \{st, cm, no\}\rangle\} \models \phi$

The following is the $\mathcal{CTR}$-S analogue of the proposition *true* of classical logic, which we denote `Playset` and define as $\phi \vee \neg\phi$.

4. CONSTRAINTS ON CONTRACT EXECUTION

In [8], we have shown how a large class of constraints on workflow execution can be expressed in $\mathcal{CTR}$. In $\mathcal{CTR}$-S we are interested in finding a similar class of constraints. In this context, verification of a constraint against a contract means that the reasoner has a way of executing the contract so that the constraint will hold

[5] Formally, the semantics of such built-ins is defined using the notion of the $\boldsymbol{transition\ oracle}$ [6].

no matter what the other parties do (for instance, that the goods are delivered or payment refunded regardless). Our verification algorithm requires that behavioral specification of contracts have no loops in them, which is captured by the *unique event property* defined below.[6]

We assume that there is a subset of propositions, $\mathcal{EVENT}$, which represents the "interesting" events in the course of workflow execution. These events are the building blocks both for workflows and constraints. In terms of Definition 3, these propositions constitute the built-in elementary updates.

Definition 4. (**Unique-event property**) A $\mathcal{CTR}$-S workflow $\mathcal{W}$ has the $\boldsymbol{unique\ event\ property}$ if and only if every proposition in $\mathcal{EVENT}$ can execute at most once in any execution of $\mathcal{W}$. Formally, this can be defined both model-theoretically and syntactically. The syntactic definition is that for every $e \in \mathcal{EVENT}$:

> If $\mathcal{W}$ is $\mathcal{W}_1 \otimes \mathcal{W}_2$ or $\mathcal{W}_1 \mid \mathcal{W}_2$ and e occurs in $\mathcal{W}_1$ then it cannot occur in $\mathcal{W}_2$, and vice versa.

For workflows with no loops, we can always rename different occurrences of the same type of event to satisfy the above property. We shall call such workflows $\boldsymbol{unique\ event\ workflows}$.

Definition 5. (**Constraints**) Let ϕ be a $\sqcap$-free formula. Then $*\phi$ denotes a formula that is true on a set of m-paths, S, if and only if ϕ is true on *every* m-path in S. The operator $*$ can be expressed using the basic machinery of $\mathcal{CTR}$-S.

Our constraints on workflow execution, defined below, will all be of the form $*\phi$, because intuitively this is the most common thing that a reasoner would want to know about the possible executions of a contract. Rules 1–3 define *primitive* constraints, denoted $\mathcal{PRIMITIVE}$. Rule 4 defines the set $\mathcal{CONSTR}$ of all constraints.

1. **Elementary primitive constraints**: If $e \in \mathcal{EVENT}$ is an event, then $*e$ and $*(\neg e)$ are primitive constraints. The constraint $*e$ is true on a set of m-paths, S, in an m-path structure $\mathbf{I} = (U, I_{\mathcal{F}}, I_{path})$ iff e occurs on every m-path in S. Similarly, $*(\neg e)$ is true on S iff e does *not* occur on any m-path in S.

   Formally, the former constraint says that every execution of the contract, *i.e.*, every m-path $\langle p_1, ..., p_i, ..., p_n\rangle \in S$, has a component-path, p_i of the form $d_1 ... d_k d_{k+1} ... d_m$, such that for some pair of adjacent states, d_k and d_{k+1}, the event e occurs at d_k and causes state transition to d_{k+1}, *i.e.*, $I_{path}(\langle d_k d_{k+1}\rangle) \models^{classic} e$ (see Definition 3). The latter constraint, $*(\neg e)$, means that e does not occur on any m-path in S.

2. **Disjunctive and Conjunctive Primitive constraints**: Any $\vee$ or $\wedge$ combination of propositions from $\mathcal{EVENT}$ is allowed under the scope of $*$.

3. **Serial primitive constraints**: If $e_1, ..., e_n \in \mathcal{EVENT}$ then $*(e_1 \otimes ... \otimes e_n)$ is a primitive constraint. It is true on a set of m-paths S iff e_1 occurs before e_2 before ... before e_n on every path in S.

4. **Complex constraints**: The set of all constraints, $\mathcal{CONSTR}$, consists of all Boolean combinations of primitive constraints using the connectives $\vee$ and $\wedge$.

[6]This assumption has good practical reasons. It is made by virtually all formal approaches to workflow modeling (*e.g.*, [2, 22]) and even such specification languages as WSFL — IBM's proposal for Web service specification language that was one of the inputs to BPEL4WS [15].

It can be shown that under the unique event assumption any serial primitive constraint can be decomposed into a conjunction of binary serial constraints. For instance, $*(e_1 \otimes e_2 \otimes e_3)$ is equivalent to $*(e_1 \otimes e_2) \wedge *(e_2 \otimes e_3)$. Here are some typical constraints in $\mathcal{CONSTR}$ and their real-world interpretation:

$*e$ – event e should always eventually happen;
$*e \wedge *f$ – events e and f must always both occur (in some order);
$*(e \vee f)$ – always either event e or event f or both must occur;
$*e \vee *f$ – either always event e must occur or always event f must occur;
$*(\neg e \vee \neg f)$ – it is not possible for e and f to happen together;
$*(e \rightarrow f)$ – if event e occurs, then f must also occur.

Example 3. (**Procurement Contract**) The contract among these actors of the procurement workflow from Example 1 can be expressed as the following constraints:

$*(approve \wedge \neg cancel \rightarrow confirm)$
if *financing* is approved *buyer* should *confirm*
$*(cancel \rightarrow recv\_escrow)$
if *buyer* cancels, then *seller* keeps the escrow
$*(make\_payment \rightarrow delivery)$
if *buyer* pays, then *seller* must deliver

5. SYNTHESIZING A COORDINATOR BY SOLVING WORKFLOW GAMES

5.1 Algorithm for Generating Strategies

As explained, given a $\mathcal{CTR}$-S workflow $\mathcal{W}$ and a winning condition Φ, our goal is to compute a $\wedge$-free workflow $\mathcal{W}_\Phi$ that is equivalent to $\mathcal{W} \wedge \Phi$. Because $\mathcal{W}_\Phi$ is $\wedge$-free, it is directly executable by the $\mathcal{CTR}$-S proof theory[7] and thus can be thought of as a set of executable strategies that the workflow coordinator can follow to enforce Φ.

Enforcing complex constraints. Let $*C_1, *C_2 \in \mathcal{CONSTR}$, $\mathcal{W}$ be a $\mathcal{CTR}$-S workflow, then

$$(*C_1 \vee *C_2) \wedge \mathcal{W} \equiv (*C_1 \wedge \mathcal{W}) \vee (*C_2 \wedge \mathcal{W})$$
$$(*C_1 \wedge *C_2) \wedge \mathcal{W} \equiv (*C_1 \wedge (*C_2 \wedge \mathcal{W}))$$

Enforcing elementary constraints. The following transformation takes an elementary primitive constraint of the form $*\alpha$ or $*\neg\alpha$ and a control flow graph (expressed as a $\mathcal{CTR}$-S unique-event workflow) and returns a $\mathcal{CTR}$-S workflow all of whose outcomes satisfy the constraint. Let $\alpha, \beta \in \mathcal{EVENT}$ and $\mathcal{W}_1, \mathcal{W}_2$ be $\mathcal{CTR}$-S workflows. Then:

$$*\alpha \wedge \alpha = \alpha \qquad *\neg\alpha \wedge \alpha = \neg\texttt{Playset}$$
$$*\alpha \wedge \beta = \neg\texttt{Playset} \qquad *\neg\alpha \wedge \beta = \beta \text{ if } \alpha \neq \beta$$

$$\begin{aligned}
\alpha \wedge (\mathcal{W}_1 \otimes \mathcal{W}_2) &= (\alpha \wedge \mathcal{W}_1) \otimes \mathcal{W}_2 \vee \mathcal{W}_1 \otimes (*\alpha \wedge \mathcal{W}_2)\\
\neg\alpha \wedge (\mathcal{W}_1 \otimes \mathcal{W}_2) &= (\neg\alpha \wedge \mathcal{W}_1) \otimes (*\neg\alpha \wedge \mathcal{W}_2)\\
\alpha \wedge (\mathcal{W}_1 \mid \mathcal{W}_2) &= (\alpha \wedge \mathcal{W}_1) \mid \mathcal{W}_2 \vee \mathcal{W}_1 \mid (*\alpha \wedge \mathcal{W}_2)\\
\neg\alpha \wedge (\mathcal{W}_1 \mid \mathcal{W}_2) &= (\neg\alpha \wedge \mathcal{W}_1) \mid (*\neg\alpha \wedge \mathcal{W}_2)
\end{aligned}$$

The above transformations are identical to those used for workflows of cooperating tasks in [8]. The first transformation below is specific to $\mathcal{CTR}$-S. Here we use σ to denote $*\alpha$ or $*\neg\alpha$:

$$\sigma \wedge (\mathcal{W}_1 \sqcap \mathcal{W}_2) = (\sigma \wedge \mathcal{W}_1) \sqcap (\sigma \wedge \mathcal{W}_2)$$
$$\sigma \wedge (\mathcal{W}_1 \vee \mathcal{W}_2) = (\sigma \wedge \mathcal{W}_1) \vee (\sigma \wedge \mathcal{W}_2)$$

[7]Not presented here due to space constraints. The proof theory can be found in http://www.cs.sunysb.edu/∼davulcu/gamectrtech.ps

For example, if $\mathcal{W} \equiv abort \sqcap prep \otimes (abort \vee commit)$, then:

$$*abort \wedge \mathcal{W} = abort \sqcap (prep \otimes abort)$$
$$*\neg abort \wedge \mathcal{W} = \neg \texttt{Playset}$$

Enforcing serial constraints. Next we extend our solver to work with constraints of the form $*(\alpha \otimes \beta)$.

Let $\alpha, \beta \in \mathcal{EVENT}$ and let $\mathcal{W}$ be a $\mathcal{CTR}$-S workflow. Then:

$$*(\alpha \otimes \beta) \wedge \mathcal{W} = \texttt{sync}(\alpha < \beta, (*\alpha \wedge (*\beta \wedge \mathcal{W})))$$

The transformation sync here is intended to synchronize events in the right order. It uses elementary updates $send(\xi)$ and $receive(\xi)$ and is defined as follows: $\texttt{sync}(\alpha < \beta, \mathcal{W}) = \mathcal{W}'$, where $\mathcal{W}'$ is like $\mathcal{W}$, except that every occurrence of event α is replaced with $\alpha \otimes send(\xi)$ and every occurrence of β with $receive(\xi) \otimes \beta$, where ξ is a new constant.

The primitives *send* and *receive* can be defined as $insert(channel(\xi))$ and $delete(channel(\xi))$, respectively, where ξ is a new symbol. The point here is that these two primitives can be used to provide synchronization: $receive(\xi)$ can become true if and only if $send(\xi)$ has been executed previously. In this way, β cannot start before α is done. The following examples illustrate this transformation:

$$*(\alpha \otimes \beta) \wedge \gamma \vee (\beta \otimes \alpha) = receive(\xi) \otimes \beta \otimes \alpha \otimes send(\xi)$$
$$*(\alpha \otimes \beta) \wedge (\alpha \mid \beta \mid \rho_1 \mid ... \mid \rho_n) =$$
$$(\alpha \otimes send(\xi)) \mid (receive(\xi) \otimes \beta) \mid \rho_1 \mid ... \mid \rho_n)$$

Proposition 1. (**Enforcing Elementary and Serial Constraints**) The above transformations for elementary and serial primitive constraints are correct, i.e., they compute a $\mathcal{CTR}$-S workflow that is equivalent to $\mathcal{W} \wedge \sigma$, where σ is an elementary or serial constraint.

Enforcing conjunctive primitive constraints. To enforce a primitive constraint of the form $*(\sigma_1 \wedge ... \wedge \sigma_m)$, where all σ_i are elementary, we utilize the logical equivalence $*(\sigma_1 \wedge ... \wedge \sigma_m) \equiv * \sigma_1 \wedge ... \wedge *\sigma_m$ (and the earlier equivalences for enforcing complex constraints).

Enforcing disjunctive primitive constraints. These constraints have the form $*(\sigma_1 \vee ... \vee \sigma_n)$ where all σ_i are *elementary constraints*. To enforce such constraints we rely on the following lemma.

Lemma 1. (**Disjunctive Primitive Constraints**) Let σ_i be elementary constraints. Then

$$*(\sigma_1 \vee ... \vee \sigma_n) \equiv (*\neg\sigma_2 \wedge ... \wedge *\neg\sigma_n \rightarrow *\sigma_1) \sqcap ...$$
$$\sqcap (*\neg\sigma_1 \wedge ... \wedge *\neg\sigma_{i-1} \wedge *\neg\sigma_{i+1} \wedge ... \wedge *\neg\sigma_n \rightarrow *\sigma_i) \sqcap ...$$
$$\sqcap (*\neg\sigma_1 \wedge ... \wedge *\neg\sigma_{n-1} \rightarrow *\sigma_n)$$

The above equivalence allows us to decompose the set of all plays in an outcome into subsets that satisfy the different implications shown in the lemma. Unfortunately, enforcing such implications is still not easy. Unlike the other constraints that we have dealt with in this section, enforcement of the implicational constraints cannot be described by a series of simple logical equivalences. Instead, we have to resort to equivalence transformations defined with the help of parse trees of workflows (expressed as $\mathcal{CTR}$-S formulas). These transformations are correct only for workflows that satisfy the unique-event property.

Definition 6. (**Maximal guarantee for an event**) Let $*\sigma$ be an elementary constraint (*i.e.*, σ is e or $\neg e$), $\mathcal{W}$ be a workflow, and φ be a subformula of $\mathcal{W}$. Then φ is said to be a $\boldsymbol{maximal\ guarantee}$ for $*\sigma$ iff

1. $(\mathcal{W} \wedge (\varphi \mid \texttt{Playset})) \rightarrow *\sigma$
2. φ is a maximal subformula of $\mathcal{W}$ that satisfies (1)

The set of all maximal guarantees for an elementary event $*\sigma$ is denoted by $GS_{*\sigma}(\mathcal{W})$. Intuitively, formulas in $GS_{*e}(\mathcal{W})$ are such that the event e is guaranteed to occur during *every* execution of such a formula. Likewise, the formulas in $GS_{*\neg e}(\mathcal{W})$ are such that the event e does not occur in *any* execution of such a formula.

Definition 7. (**Co-executing sub-games of a subformula**) Let $\mathcal{W}$ be a workflow and ψ, φ be a pair of subformulas of $\mathcal{W}$. Then ψ and φ $\boldsymbol{coexecute}$ in $\mathcal{W}$, denoted
$\psi \in coExec(\mathcal{W}, \varphi)$ iff

1. $(\mathcal{W} \wedge (\varphi \mid \texttt{Playset})) \rightarrow (\psi \mid \texttt{Playset})$,
2. ϕ and ψ are disjoint subformulas in $\mathcal{W}$, and
3. ψ is a maximal subformula in $\mathcal{W}$ satisfying (1) and (2)

Intuitively members of $coExec(\mathcal{W}, \varphi)$ correspond to maximal sub-games of $\mathcal{W}$ that must execute whenever φ executes as part of the workflow $\mathcal{W}$.

Proposition 2. (**Computing $GS_{*\sigma}(\mathcal{W})$ and $coExec(\mathcal{W}, \varphi)$**) The following procedures compute $GS_{*\sigma}(\mathcal{W})$ and $coExec(\mathcal{W}, \varphi)$. They operate on the parse tree of $\mathcal{W}$, which is defined as usual: the inner nodes correspond to composite subformulas and the leaves to atomic subformulas. Thus, the leaves are labeled with their corresponding atomic subformulas, while the inner nodes are labeled with the main connective of the corresponding composite subformula.

$GS_{*e}(\mathcal{W})$**:** The set of subformulas that correspond to the nodes in the parse tree of $\mathcal{W}$ that are closest to the root of the tree and can be reached by the following marking procedure: (i) mark all the leaves labeled with e; (ii) mark any node labeled with a subformula of $\mathcal{W}$ of the form $(\varphi \otimes \psi)$ or $(\varphi \mid \psi)$ if either the node for φ or that for ψ is marked; (iii) mark any node labeled with a subformula of the form $(\varphi \vee \psi)$ or $(\varphi \sqcap \psi)$ if both the node for φ *and* the node for ψ are marked.

$coExec(\mathcal{W}, \varphi)$**:** Find all subformulas of $\mathcal{W}$ of the form $\psi_1 \mid \psi_2$ or $\psi_1 \otimes \psi_2$ that contain φ as a subformula. Then, $\psi_1 \in coExec(\mathcal{W}, \varphi)$ if φ is a subformula of ψ_2 and ψ_1 is a maximal subformula of $\mathcal{W}$ with this property. Likewise $\psi_2 \in coExec(\mathcal{W}, \varphi)$ if φ is a subformula of ψ_1 and ψ_2 is maximal.

$GS_{*\neg e}(\mathcal{W})$**:** Let T be the set of nodes in the parse tree of $\mathcal{W}$ that belong to any of the subformulas $\varphi \in GS_{*e}(\mathcal{W})$ or $\psi \in coExec(\mathcal{W}, \varphi)$. Then, $\eta \in GS_{*\neg e}(\mathcal{W})$ iff it is a subformula of $\mathcal{W}$ such that its subtree contains no nodes in T and η is a maximal subformula with such a property.

Example 4. (**Computation of maximal guarantees and co-execution**) The workflow parse tree for Example 3 is shown in Figure 1. For the event $approve$, the set $GS_{*approve}(\mathcal{W})$ is $\{approve \otimes (makepay \vee cancel)\}$. It contains a single maximal guarantee subformula for $approve$, which corresponds to the node labeled $n1$ in the figure. The set of co-executing subformulas for $n1$, $coExec(\mathcal{W}, n1)$, consists of two formulas that correspond to the nodes $n3$ and $n4$ in the figure. The only maximal guarantee for $\neg approve$ is the subformula $rej \otimes cancel$, which corresponds to node $n2$.

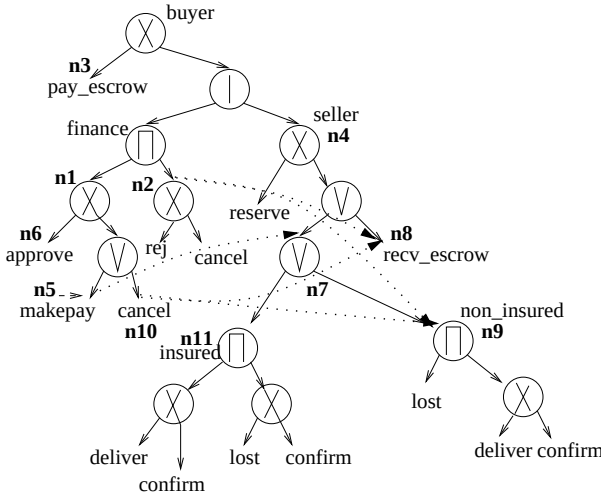

Figure 1: Workflow parse tree and workflow graph for Example 3

procedure $solve((*\sigma_1 \wedge ... \wedge *\sigma_n \rightarrow *\sigma),\ \mathcal{W})$
1. **if** $\forall\, i,\ \mathcal{W} \models *\sigma_i$ **then compute** $\mathcal{W} \wedge *\sigma$
2. **else** $Guard(g) := \emptyset$ **for all** $g \in subformulas(\mathcal{W})$
3. **for each** i **such that** $\mathcal{W} \not\models *\sigma_i$ **do**
4. **for each** $f \in GS_{*\neg\sigma_i}(\mathcal{W})$ **do**
5. **if** $\exists h \in coExec(\mathcal{W}, f)$ **and** $(h \wedge *\sigma)$ **is satisfiable**
6. **then rewrite** f **to** $send(\xi) \otimes f$ **and set**
7. $Guard(g) := Guard(g) \cup \{recv(\xi)\},$
8. $\forall g \in GS_{*\neg\sigma}(h)$
9. **else** $solve((*\sigma_1 \wedge ... \wedge *\sigma_n \rightarrow *\sigma),\ sibling(f))$
10. **for each** $g \in subformulas(\mathcal{W})$ **do**
11. **if** $Guard(g) \neq \emptyset$ **then**
12. rewrite g to $\big(\bigvee_{recv(\xi) \in Guard(g)} recv(\xi)\big) \otimes g$

Figure 2: Enforcement of $((*\sigma_1 \wedge ... \wedge *\sigma_n \rightarrow *\sigma)\ \wedge\ \mathcal{W})$

The algorithm in Figure 2 computes a workflow that is equivalent to $(*\sigma_1 \wedge ... \wedge *\sigma_n \rightarrow *\sigma)\ \wedge\ \mathcal{W}$ (and thus enforces the constraint on $\mathcal{W}$), where $*\sigma_i, *\sigma$ are all elementary constraints. If the precondition of the constraint is true whenever the game is played then (in line 1) $*\sigma$ is enforced on $\mathcal{W}$. Otherwise, for every $*\sigma_i$ that is not true everywhere, we identify the subgames $f \in GS_{*\neg\sigma_i}$ (lines 2-3). Note that, whenever subgames in $GS_{*\neg\sigma_i}$ are executed the constraint $*\sigma_1 \wedge ... \wedge *\sigma_n \rightarrow *\sigma$ is vacuously true. In lines 4-6, we identify the subformulas h of $\mathcal{W}$ that co-execute with the formulas $f \in GS_{*\neg\sigma_i}$. If $*\sigma$ is enforcible in any of these subformulas h, *i.e.*, $h \wedge *\sigma$ is satisfied (there can be at most one such subformula h per f, due to the unique event property, Definition 4), then we enforce the above constraint by *delaying* executions of those subgames in h that violate $*\sigma$ (these are exactly the g's in line 7) *until* it is guaranteed that the game moves into $f \in GS_{*\neg\sigma_i}$, because once f is executed our constraint becomes satisfied. This delay is achieved by synchronizing the executions of f to occur before the executions of g by rewriting f into $send(\xi) \otimes f$ and by adding $recv(\xi)$ to the guard of g (in line 6). Otherwise, if no such h exists, in line 9, we explicitly enforce the constraint on the sibling nodes (in the parse tree of $\mathcal{W}$) of the formulas $f \in GS_{*\neg\sigma_i}$ (because an outcome that satisfies $*\sigma_i$ might exist in a sibling). Finally, in lines 10-12, we make sure that the execution of every g that has a non-empty guard is conditioned on receiving of a message from at least one f with which g is synchronized.

Example 5. (**Procurement Workflow – contd.**) The algorithm in Figure 2 creates the following workflow by applying the constraints in Example 3 to the procurement workflow in Example 1. The parse tree for that workflow is depicted in Figure 1.

- To enforce $(*cancel \rightarrow *recv\_escrow)$ we first compute $GS_{*\neg cancel}(buyer) = \{n5\}$, and $coExec(buyer, n5) = \{n3, n4, n6\}$. Of these, $n4$ (substituted for h) satisfies the conditions of line 5 of the algorithm in Figure 2. Since $GS_{*\neg recv\_escrow}(n4) = \{n7\}$, we insert a synchronization from node $n5$ to $n7$ shown in Figure 1 as a dotted line. This ensures that if the buyer cancels the procurement workflow, the seller collects the escrow.

- To enforce $(*approve \wedge\ *\neg cancel \rightarrow *confirm)$, we compute $GS_{*\neg approve}(buyer) = \{n2\}$ and notice that $n4 \in coExec(buyer, n2)\}$ satisfies the conditions in line 5 of the algorithm in Figure 2. Since $GS_{*\neg confirm}(n4) = \{n8, n9\}$, we insert a synchronization from node $n2$ to $n8$ and $n9$ which yields the dotted edges in Figure 1. We also compute $GS_{*cancel}(buyer) = \{n10, n2\}$ and notice that $n4 \in coExec(buyer, n2)\}$, $n4 \in coExec(buyer, n10)\}$, and $n4$ satisfies the conditions in line 5 of the algorithm. Since $GS_{*\neg confirm}(n4) = \{n8, n9\}$, we insert a synchronization from the nodes $n10$ and $n2$ to $n8$ and $n9$, which yields the dotted edges in Figure 1. This synchronization ensures that if buyer's financing is approved and he chooses to make the payment and buy the item then delivery must use the insured method. Also, once the constraint $(*make\_payment \rightarrow *delivery)$ is enforced too, the seller can no longer pocket the escrow. The resulting strategy is:

$$
\begin{aligned}
buyer &\longleftarrow pay\_escrow \otimes (financing \mid seller)\\
financing &\longleftarrow (approve \otimes ((send(\xi_1) \otimes make\_payment) \vee\\
&((send(\xi_3) \otimes cancel))) \sqcap (send(\xi_2) \otimes (reject \otimes cancel))\\
seller &\longleftarrow reserve\_item \otimes ((recv(\xi_1) \otimes delivery)\\
&\quad \vee ((recv(\xi_2) \vee recv(\xi_3)) \otimes collect\_escrow))\\
delivery &\longleftarrow insured \vee ((recv(\xi_2) \vee recv(\xi_3)) \otimes non\_insured)\\
insured &\longleftarrow (delivered \otimes confirm) \sqcap (lost \otimes confirm)\\
non\_insured &\longleftarrow (delivered \otimes confirm) \sqcap lost
\end{aligned}
$$

Proposition 3. (**Enforcing disjunctive primitive constraints**) The above algorithm for enforcing disjunctive primitive constraints is correct, i.e., it computes a $\mathcal{CTR}$-S workflow that is equivalent to $\mathcal{W} \wedge *(\sigma_1 \vee ... \vee \sigma_n)$ where σ_i are *elementary constraints*.

THEOREM 1 (COMPLEXITY OF ENFORCING DPC). *Let $\mathcal{W}$ be a control flow graph and $*\Phi \in \mathcal{PRIMITIVES}$ be a disjunctive primitive constraint(DPC). Let $|\mathcal{W}|$ denote the size of $\mathcal{W}$, and d be the number of elementary disjuncts in $*\Phi$. Then the worst-case size of $*\Phi \wedge \mathcal{W}$ is $O(d \times |\mathcal{W}|)$, and the time complexity is $O(d \times |\mathcal{W}|^2)$*

THEOREM 2 (COMPLEXITY OF SOLVING GAMES). *Let $\mathcal{W}$ be a control flow graph $\mathcal{W}$ and $\Phi \subset \mathcal{CONSTR}$ be a set of global constraints in the conjunctive normal form $\bigwedge_N(\vee_j Prim)$ where $Prim \in \mathcal{PRIMITIVES}$. Let $|\mathcal{W}|$ denote the size of $\mathcal{W}$, N be the number of constraints in Φ, and d be the largest number of disjuncts in a primitive constraint in Φ. Then the worst-case size of $solve(\Phi, \mathcal{W}) \doteq \Phi \wedge \mathcal{W}$ is $O(d^N \times |\mathcal{W}|)$, and the time complexity is $O(d^N \times |\mathcal{W}|^2)$.*

Cycle detection and removal. After compiling the constraints Φ into the workflow $\mathcal{W}$, several things still need to be done. The problem is that even though $\mathcal{W}_\Phi$ is an executable workflow specification, $\mathcal{W}_\Phi$ may have sub-formulas where the *send/receive* primitives cause a cyclic wait, which we call *cyclic blocks*. Fortunately, we can show that a variant of depth first search on the control flow graph of $\mathcal{W}$ can identify all executable cyclic blocks and remove all blocks from $\mathcal{W}$ in time $O(|\mathcal{W}|^3)$.

Example 6. **(Cyclic Block Removal)** Let $\mathcal{W} \leftarrow (a \vee b) \otimes (c \sqcap d)$ and C be $(*c \rightarrow *a)$. Our algorithm will compile $\mathcal{W} \wedge C$ into $(a \vee recv(\overline{\xi}) \otimes b) \otimes (c \sqcap (send(\overline{\xi}) \otimes d))$. Now, if `reasoner` moves into b, a deadlock occurs. However, we can rewrite this workflow into $a \otimes (c \sqcap d)$ to avoid the problem.

6. CONCLUSION AND RELATED WORK

We presented a novel formalism, $\mathcal{CTR}$-S, for modeling the dynamics of service contracts. $\mathcal{CTR}$-S is a logic in which service contracts are represented as formulas that specify the various choices that are allowed for the parties to the contracts. The logic permits the reasoner to state the desired outcomes of the contract execution and verify that a desired outcome can be achived no matter what the other parties do as long as they obey the rules of the contract.

There is a body of preliminary work trying to formalize the representation of Web service contracts [20, 11], but none deals with the dynamics of such contracts, which is the main subject of this paper. Technically, the works closest to ours come from the fields of model checking and game logics.

Process algebras and alternating temporal logic [7, 1] have been used for modeling open systems with game semantics. Model checking is a standard mechanism for verifying temporal properties of such systems and deriving automata for scheduling. In [16], the complexity and size of computing the winning strategies for infinite games played on finite graphs are explored. A result analogous to ours is obtained for infinite games: assuming the size of the graph is Q and the size of the winning condition is W the complexity of computing winning strategies is exponential in the size of W and polynomial in the size of the set Q.

The use of $\mathcal{CTR}$-S has enabled us to find a more efficient verification algorithm than what one would get using model checking. Indeed, standard model checking techniques [7, 1] are worst-case exponential *in the size of the entire control flow graph* and the corresponding scheduling automata are also exponential. This is often referred to as the *state-explosion problem*. In contrast, the size of our solver is *linear* in the size of the original workflow graph and exponential only in the size of the constraint set (Theorem 2), which is a much smaller object. In a sense, our solver can be viewed as a specialized and more efficient model checker/scheduler for the problem at hand. It accepts high level specifications of workflows and yields strategies and schedulers in the same high level language.

Logic games have been proposed before in other contexts [13, 19]. As in $\mathcal{CTR}$-S, validity of a statement in such a logic means that the reasoner has a winning strategy against the opponent. In $\mathcal{CTR}$-S however, games, winning conditions, and strategies are *themselves* logical formulas (rather than modal operators). Logical equivalence in $\mathcal{CTR}$-S is a basis for constructive algorithms for solving games and synthesizing strategies, which are in turn executable by the proof theory of $\mathcal{CTR}$-S. Related game logic formalisms, such as [13, 19], only deal with assertions about games and their winning strategies. In these logics, games are modalities rather than executable specifications, so they can only be used for *reasoning* about Web service contracts, but *not* for *modeling* and *executing* them.

Related work in planning, where planning goals are expressed as temporal formulas includes [3]. In [3], plans are generated using a forward chaining engine that generates finite linear sequences of actions. As these linear sequences are generated, the paths are incrementally checked against the temporal goals. This approach is sound and complete. However, in the worst case it performs an exhaustive search of the model similar to the model checking approaches.

For the future work, we are planning to extend our results to allow contracts that include iterative behaviour. Such contracts can already be specified in $\mathcal{CTR}$-S. However, iteration requires new verification algorithms to enable reasoning about the desired outcomes of such contracts.

7. REFERENCES

[1] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. In *Intl. Conference on Foundations of Computer Science*, pages 100–109, 1997.

[2] P.C. Attie, M.P. Singh, E.A. Emerson, A.P. Sheth, and M. Rusinkiewicz. Scheduling workflows by enforcing intertask dependencies. *Distributed Systems Engineering Journal*, 3(4):222–238, December 1996.

[3] Fahiem Bacchus and Froduald Kabanza. Planning for temporally extended goals. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 1215–1222, Portland, Oregon, USA, 1996. AAAI Press / The MIT Press.

[4] A.J. Bonner. Workflow, transactions, and datalog. In *ACM Symposium on Principles of Database Systems*, Philadelphia, PA, May/June 1999.

[5] A.J. Bonner and M. Kifer. Concurrency and communication in transaction logic. In *Joint Intl. Conference and Symposium on Logic Programming*, pages 142–156, Bonn, Germany, September 1996. MIT Press.

[6] A.J. Bonner and M. Kifer. A logic for programming database transactions. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, chapter 5, pages 117–166. Kluwer Academic Publishers, March 1998.

[7] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, pages 244–263, 1986.

[8] H. Davulcu, M. Kifer, C.R. Ramakrishnan, and I.V. Ramakrishnan. Logic based modeling and analysis of workflows. In *ACM Symposium on Principles of Database Systems*, pages 25–33, Seattle, Washington, June 1998.

[9] E.A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, pages 997–1072. Elsevier and MIT Press, 1990.

[10] R. Fagin, J. Y. Halpern, Y. Moses, and Moshe Y. Vardi. *Reasoning About Knowledge*. MIT Press, 1994.

[11] B.N. Grosof and T.C. Poon. Sweetdeal: representing agent contracts with exceptions using xml rules, ontologies, and process descriptions. In *Proceedings of the twelfth international conference on World Wide Web*, pages 340–349, May 2003.

[12] R. Gunthor. Extended transaction processing based on dependency rules. In *Proceedings of the RIDE-IMS Workshop*, 1993.

[13] J. Hintikka. *Logic, Language Games, and Information*. Oxford Univ. Press, Clarendon, Oxford, 1973.

[14] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of ACM*, pages 741–843, July 1995.

[15] F. Leymann. Web services flow language (wsfl 1.0). Technical report, IBM, 2001. http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf.

[16] R. McNaughton. Infinite games played on finite graphs. *Annals of Pure and Applied Logic*, 65:149–184, 1993.

[17] S. Mukherjee, H. Davulcu, M. Kifer, G. Yang, and P. Senkul. Survey of logic based approaches to workflow modeling. In J. Chomicki, R. van der Meyden, and G. Saake Springer, editors, *Logics for Emerging Applications of Databases*, LNCS. Springer-Verlag, October 2004.

[18] M. Osborne and A. Rubinstein. *A Course in Game Theory*. The MIT Press, 1998.

[19] Rohit Parikh. Logic of games and its applications. In *Annals of Discrete Mathematics*, volume 24, pages 111–140. Elsevier Science Publishers, March 1985.

[20] D.M. Reeves, M.P. Wellman, and B.N. Grosof. Automated negotiation from declarative contract descriptions. In J.P. Müller, E. Andre, S. Sen, and C. Frasson, editors, *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 51–58, Montreal, Canada, 2001. ACM Press.

[21] C. Schlenoff, M. Gruninger, M. Ciocoiu, and J. Lee. The essence of the process specification language. *Transactions of the Society for Computer Simulation International*, 16(4):204–216, 2000.

[22] M.P. Singh. Semantical considerations on workflows: An algebra for intertask dependencies. In *Proceedings of the International Workshop on Database Programming Languages*, Gubbio, Umbria, Italy, September 6–8 1995.

[23] M.P. Singh. Synthesizing distributed constrained events from transactional workflow specifications. In *Proceedings of 12-th IEEE Intl. Conference on Data Engineering*, pages 616–623, New Orleans, LA, February 1996.