A Game Logic for Workflows of Non-cooperating Services *

Hasan Davulcu Michael Kifer I.V. Ramakrishnan
Department of Computer Science
Stony Brook University
Stony Brook, NY 11794
{davulcu, kifer, ram}@cs.stonybrook.edu

Abstract

A *workflow* is a collection of coordinated activities designed to achieve a well-defined common goal. Extant research on modeling and analysis of workflows by and large assume that a workflow consists entirely of *co-operating* tasks that are all controlled by the workflow scheduler. However in real life – especially in the emerging field of Web services, workflows operate in an *open* environment where some of the tasks they interact with are not under the control of the workflow manager. Indeed, these tasks may be not only non-cooperative but even adversarial.

Reasoning about workflows in the presence of such tasks is an unexplored topic. In this paper we develop a framework to reason about them. Our formalism, called Game-CTR, expands our previous work on workflow modeling and reasoning using Concurrent Transaction Logic (CTR) [7] with concepts from Game Theory. We develop a model and proof theory for Game-CTR and demonstrate its applicability for modeling, reasoning, and co-ordinating tasks in workflows that operate in open environments. Game-CTR is a natural generalization of CTR. A pleasant consequence of this generalization is that the results in our previous work [13] emerge as special cases.

Contact author : Hasan Davulcu
E-mail : davulcu@cs.stonybrook.edu
Phone : (631) 689-1631
Fax : (631) 632-8334

*This work was supported in part by NSF grants INT98-09945 and IIS-0072927.

1 Introduction

A *workflow* [14] is a collection of coordinated activities designed to achieve a well-defined common goal. An important problem in workflow management is *scheduling* of workflow tasks in ways that satisfy various temporal and causality constraints. Extant research on modeling and analysis of workflows by and large assume that a workflow consists entirely of *co-operating* tasks that are all controlled by the workflow scheduler [13, 2, 3, 9, 15, 20, 25]. However, in real-life — especially in the emerging field of Web services [23, 11] —workflows operate in an *open* environment (such as servers or external software systems) where some of the tasks are not under the control of one workflow scheduler or even one organization. A simple scenario could be one where a buyer interacts with an external seller and delivery service, and buyer needs to be assured that the goods will either be delivered 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 the external services might not only be non-cooperative but also adversarial to an extent. Reasoning about workflows in the presence of such services is an unexplored research area and constitutes the topic of this paper.

In our earlier work [13], we showed that Concurrent Transaction Logic (abbr., $\mathcal{CTR}$) [7] is a natural logical formalism for representing workflow control graphs, temporal and causality constraints on workflow execution, for reasoning about the consistency of workflow specifications, and for scheduling workflows in the presence of those constraints. This work assumed that the workflow consists entirely of cooperating tasks. In this paper we substantially expand our earlier work for reasoning about workflows that operate in open environments. Specifically we develop Game-$\mathcal{CTR}$, an extension of $\mathcal{CTR}$, as the underlying formal framework for designing, modeling and reasoning about run-time properties of workflows that are composed of non-cooperating services.

Overview and summary of results. The idea is to combine $\mathcal{CTR}$ with certain concepts from Game Theory [21, 16, 22]. Game-$\mathcal{CTR}$ extends $\mathcal{CTR}$ with one new connective — the *opponent's conjunction* — which represents an opponent's choice of action that might not be in consistent with the overall goal of the workflow. To accommodate the new connective we develop a model theory for Game-$\mathcal{CTR}$ and show how it can be used to specify executions under a fairly large class of temporal and causality constraints.

The Game-$\mathcal{CTR}$ model theory characterizes all possible *outcomes* of a game formula $\mathcal{G}$ that represents a workflow. A constraint $\mathcal{C}$ in Game-$\mathcal{CTR}$ acts as a *winning condition* for a game; it characterizes outcomes with certain desirable properties. The formula $\mathcal{G} \wedge \mathcal{C}$ characterizes those outcomes that satisfy the constraint $\mathcal{C}$. If $\mathcal{G} \wedge \mathcal{C}$ is satisfiable, meaning that there exists at least an outcome $\mathcal{O}$ in its model, then we say that the constraint $\mathcal{C}$ is *enforcible* in game $\mathcal{G}$.

We describe an algorithm that converts specifications such as $\mathcal{G} \wedge \mathcal{C}$ into other, equivalent Game-$\mathcal{CTR}$ formulas that can be executed more efficiently and without backtracking. The transformation also flags unsatisfiable workflow specifications. In a well-defined sense, the result of such a transformation can be thought of as a concise representation of all *winning strategies* of the game that the workflow scheduler plays against the adversary part of the workflow.

Thus, in Game-$\mathcal{CTR}$, familiar workflow concepts have game-theoretic interpretation. For instance, the workflow *scheduler* corresponds to `player`; non-cooperating autonomous *task agents* collectively correspond to `opponent`, the workflow control structure corresponds to the rules *of the game*, and *task dependencies* correspond to the *winning conditions*. Both the rules of the game, $\mathcal{G}$, and the winning conditions, $\mathcal{C}$, are represented as Game-$\mathcal{CTR}$ formulas.

Finally, since Game-$\mathcal{CTR}$ is a natural generalization of CTR, a pleasing aspect of this work is that our earlier results in [13] becomes special cases of the results in this paper.

The rest of the paper is organized as follows. In Section 2, we introduce Game-$\mathcal{CTR}$ and discuss workflow modeling in this logic. In Section 3, we introduce the model theory of Game-$\mathcal{CTR}$. The proof theory of Game-$\mathcal{CTR}$ is discussed in Section 4. Section 5 presents the causal and temporal workflow constraints. In Section 6, we present the game solver algorithm and its complexity. In Section 6, we discuss related formalisms.

2 Game-$\mathcal{CTR}$ and Workflows

Understanding of Game-$\mathcal{CTR}$ and its relationship to workflows requires understanding of our prior work on $\mathcal{CTR}$ [7] and workflow modeling [12]. We describe the syntax of Game-$\mathcal{CTR}$ (and that of $\mathcal{CTR}$, since the latter is missing only one connective, $\sqcap$) and then provide the intuition to help the reader make sense out of the formal definitions. The model theory of Game-$\mathcal{CTR}$ is a rather radical extension of that of $\mathcal{CTR}$ and it requires getting used to. It is described in Section 3.

2.1 Syntax

The atomic formulas of Game-$\mathcal{CTR}$ 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 Game-$\mathcal{CTR}$ 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, and $\otimes$ is called ***serial conjunction***. The formula $\phi \mid \psi$ denotes an interleaved execute of two games ϕ and ψ, and $\mid$ is called ***concurrent conjunction***. The formula $\phi \sqcap \psi$ means that `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 `player` makes the decision. In $\mathcal{CTR}$ this connective is called ***classical disjunction*** but because of its interpretation as player's choice we will also refer to it as ***player's disjunction***. Finally, the formula $\phi \wedge \psi$ means denotes execution of ϕ *constrained* by ψ (or vice versa). It is called ***classical conjunction***.[1]

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

Definition 2.1 *(Workflow Control Specification).* A *workflow control specification* is a formula of the following form:

***Atomic*:** An atomic formula

Game-$\mathcal{CTR}$ *goal*: If ϕ and ψ are Game-$\mathcal{CTR}$ workflows then so are $\phi \otimes \psi$, $\phi \mid \psi$, $\phi \vee \psi$, and $\phi \sqcap \psi$

***Rule*:** A rule of the form $head \leftarrow body$, where *head* is an atomic formula and *body* a Game-$\mathcal{CTR}$ goal.

In this paper we will restrict workflows to be non-recursive, so the inclusion of the rules is just a matter of convenience, which does not affect the expressive power. Note also that some connectives, like $\&$ are not allowed in workflow control specifications, but they will be essential for expressing workflow constraints. To save space, we omit a number of other connectives of $\mathcal{CTR}$ (*e.g.*, isolation, $\odot$, possibility, $\diamond$) which, while useful for workflow specification, are not essential for understanding our results.

2.2 Modeling Workflows with $\mathcal{CTR}$ and Game-$\mathcal{CTR}$

Example 2.2 *(Database Transactions).* Consider an application that consists of a concurrent execution of a classical local transaction and a distributed two-phase commit transaction. Transactions are modeled in terms of their *significant events*. Thus, the classical transaction begins when the significant event $start_c$ occurs. Then it can either $abort_c$ or $commit_c$, which is followed by either compensation or a no-op. Thus, at a high level, this transaction can be represented as $start_c \otimes$ *abort-or-commit*$_c \otimes$ compensate-or-noop$_c$.

The connective $\otimes$ represents succession of events or actions. For instance, when the above expression "executes," the underlying database state changes first to one where the transaction has started (which would typically be expressed by insertion of an appropriate fact into a log), then to one where either an abort or a commit event happened (which, for now, is represented by the event *abort-or-commit*$_c$) and then to one where the transaction was either compensated or no action was taken.

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 ends up in the final state of the sequence when the execution terminates. [2]

The distributed transaction is represented similarly. After starting, it can be either aborted by itself or it would execute the two-phase commit protocol. We represent this as $start_d \otimes$ *twophase-or-abort*$_d$.

Since the overall application consists of a concurrent execution the classical and the two-phase transaction, we can represent it as a formula ϕ of the form

$$(start_c \otimes \textit{abort-or-commit}_c \otimes \textit{compensate-or-noop}_c) \mid (start_d \otimes \textit{twophase-or-abort}_d)$$

[1] 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.

[2] 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 [7, 8].

The connective $|$ represents concurrent, *interleaved* execution of the two sequences of actions. Intuitively, this means that a legal execution of ϕ is a sequence of database states where the initial subsequence corresponds, say, to a partial execution of the subformula to the left of $|$; the next subsequence of states corresponds to the execution of the right subformula; the following subsequence is a continuation of the execution of the left subformula; and so on. The overall execution sequence of ϕ is a merge of an execution sequence for the left subformula and an execution sequence for the right subformula.

Next we would like to drill-down into the composite events such as *abort-or-commit*$_c$ and *compensate-or-noop*$_c$, which is where the distinction between $\mathcal{CTR}$ and Game-$\mathcal{CTR}$ should become clear. In $\mathcal{CTR}$, these events would be represented via the following two formulas:

$$abort_c \vee commit_c \qquad \text{and} \qquad compensate_c \vee \textit{noop}_c$$

The connective $\vee$ here represents alternative execution. For instance, a legal execution of $abort_c \vee commit_c$ is either a legal execution of $abort_c$ or of $commit_c$. It can even be a sequence of states that is a legal execution of *both* (but such sequences exist rarely).

Thus, $\vee$ represents a *choice*. The question is, however, whose choice it is. In our earlier work on workflow modeling [12], the choice was always made by the workflow scheduler, so $\mathcal{CTR}$'s $\vee$ was sufficient. However, in an environment where workflow activities might not always cooperate, we need a way to represent the choice made by the "opponent." For instance, if our classical transaction represents delivery of goods by an e-Bay merchant, the goods might not be delivered, even if we paid for them, if the merchant decides to abort the transaction. Thus, in our example, the choice of whether to commit or abort belongs to the "opponent," while the choice of what to do afterwards (*i.e.*, whether to use compensation or do nothing) is made by "us," the workflow scheduler. Therefore, the full specification of the classical transaction is now

$$start_c \otimes (abort_c \sqcap commit_c) \otimes (compensate_c \vee \textit{noop}_c)$$

where the connective $\sqcap$ represents the opponent's choice.

The specification of the distributed transaction is more complex. The two-phase part of the transaction is $start_d \otimes (commit_d \vee abort_d)$, meaning that (according to the two-phase commit protocol) the scheduler if free to commit or abort the transaction if it is prepared to commit. At the same time, rather than preparing to commit, the transaction (which is scheduler's opponent) is free to decide to abort. Thus, the complete specification of the distributed transaction is

$$start_d \otimes (abort_d \sqcap (prepare_d \otimes (commit_d \vee abort_d)))$$

These unwieldy 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 Game-$\mathcal{CTR}$ goal, is that *head* is a "procedure name" for *body*. 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 [8]. With this in mind, we can now express the above workflow as follows:

$$\begin{array}{l} global\_transaction \leftarrow distributed \mid classical \\ classical \leftarrow start_c \otimes (abort_c \sqcap commit_c) \otimes (compensate_c \vee \text{noop}_c) \\ distributed \leftarrow start_d \otimes (twophase_d \sqcap abort_d) \\ twophase_d \leftarrow prepare_d \otimes (commit_d \vee abort_d) \end{array} \tag{2.1}$$

If the above specifications are viewed as the rules of a game in which the scheduler plays against independence-minded transactions, what are the "winning conditions?" In our setting, the winning conditions are constraints on the execution of the entire workflow. For the scheduler, "winning" means making sure that the execution always ends up satisfying the constraints. In our case, the scheduler might want to ensure that if *classical* aborts then so should *distributed*. Furthermore, when classical transaction commits, then $noop_c$ should execute and $compensate_c$ otherwise.

It is easy to see that regardless of what the transactions do the coordinator can make sure (by playing the appropriate branches of the $\vee$-subformulas) that the above conditions are satisfied by the execution. Thus, the scheduler has a *strategy* to win and the above constraints are *enforcible*.

We shall see that a large class of temporal and causality constraints (including those in our example) can be represented as Game-$\mathcal{CTR}$ formulas. If C 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 $global\_transaction \wedge C$ is satisfiable in Game-$\mathcal{CTR}$. In this way, Game-$\mathcal{CTR}$ provides logical account for a class of 2-player games and a modeling tool for workflows composed of independent, non-cooperating tasks.

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 workflows with non-cooperating tasks and serves as the yardstick of correctness for the algorithms in Section 6.

3.1 Multipath Sets

The semantics of Transaction Logic [5, 6] is based on sequences of database states, $p = d_1 ... d_n$, called $\boldsymbol{paths}$. Informally, a $\boldsymbol{database\ state}$ can be a collection of facts or even deductive rules, but for this paper we can think of them as simply symbolic identifiers for various collections of facts.

$\mathcal{CTR}$ [7] extends Transaction Logic with concurrent, interleaved execution, and its 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 $\boldsymbol{m\text{-}path}$ [7]. 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. As explained in Example 2.2, multipaths capture the idea of an execution of a transaction, which may be broken into segments, such that other transactions might execute in-between the segments.

Game-$\mathcal{CTR}$ further extends this idea by recognizing that in the presence of opponent's choice, the player cannot be certain which execution (or "play") will actually take place, due to the uncertainty regarding the opponent's moves. However, the player might 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, Game-$\mathcal{CTR}$ 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 Game-$\mathcal{CTR}$ 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.

3.2 Satisfaction on Multipath Sets

The following definitions make the above discussion precise.

Definition 3.1 *(m-Path Structures).* An $\boldsymbol{m\text{-}path\ structure}$ [7] 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 classical first-order semantic structure.

Given 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. The actual definitions can be found in [7]. 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.

The present paper extends this notion for Game-$\mathcal{CTR}$ 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 that of $\mathcal{CTR}$ for formulas that have no $\sqcap$'s.

Definition 3.2 *(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. 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, $p(t_1, ..., t_n)$ would either be defined via rules (as in Example 2.2) or would be a "built-in," such as $insert(q(a, b))$, with a fixed meaning. For instance, in case of $insert(q(a, b))$ this meaning would be that $\mathbf{I}, \{\pi\} \models_\nu insert(q(a, b))$ iff π is a path of the form $\langle d_1, d_2 \rangle$, where $d_2 = d_1 \cup \{q(a, b)\}$.[3] 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 2.2, 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] Formally, the semantics of such built-ins is defined using the notion of the $\boldsymbol{transition\ oracle}$ [6, 8].

3. **Player'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, $\phi \wedge \psi$, 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, $\sqcup$, also exists, but is not needed in this paper.

5. **Serial Conjunction:** $\mathbf{I}, S \models_\nu \phi \otimes \psi$ if and only if there is a set of m-paths, R, 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 of m-paths R 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 3.3 *(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 3.2: $\{\langle d\ d \cup \{st\}\rangle\} \models insert(st)$, and $\{\langle d\ d \cup \{ab\}\rangle\} \models insert(ab)$, and $\{\langle d\ d \cup \{cm\}\rangle\} \models insert(cm)$, and $\{\langle d\ d \cup \{cp\}\rangle\} \models insert(cp)$, and $\{\langle d\ d \cup \{no\}\rangle\} \models insert(no)$
2. By (3) in Definition 3.2: $\{\langle d\ d \cup \{cp\}\rangle\} \models (insert(cp) \vee insert(no))$, and $\{\langle d\ d \cup \{no\}\rangle\} \models (insert(cp) \vee insert(no))$
3. By (5) in Definition 3.2: $\{\langle d\ d \cup \{ab\}\rangle,\ \langle d\ d \cup \{cm\}\rangle\} \models (insert(ab) \sqcap insert(cm))$
4. By (6) in Definition 3.2: $\{\langle \emptyset\ \{st\}\ \{st, ab\}\rangle,\ \langle \emptyset\ \{st\}\ \{st, cm\}\rangle\} \models insert(st) \otimes (insert(ab) \sqcap insert(cm))$
5. By (6) in Definition 3.2: $\{\langle \emptyset\ \{st\}\ \{st, ab\}, \{st, ab, cp\}\rangle,\ \langle \emptyset \{st\} \{st, cm\} \{st, cm, cp\}\rangle\} \models \phi$, 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 Game-$\mathcal{CTR}$ analogue of the proposition *true* of classical logic, which we denote `Playset` and define as $\phi \vee \neg\phi$. It is easy to see that for any m-path structure, $\mathbf{I}$, and any set of m-paths, S, $\mathbf{I}, S \models$ `Playset` and that $\neg$`Playset` is unsatisfiable. Intuitively, if `Playset` is the game, then any outcome is possible, *i.e.*, and the player can force the game towards any outcome.[4] In contrast, $\neg$`Playset` means that no outcomes exist, so the opponent can deny the player any strategy whatsoever. The following are some related tautologies: $\phi \sqcap \neg$`Playset` $\equiv \neg$`Playset` and $\phi \sqcap$ `Playset` $\leftarrow \phi$.

[4] Recall that an "outcome" is the set of plays that the *player*, not the opponent, can force the game to.

4 Proof Theory

Like classical logic, Game-$\mathcal{CTR}$ has a "Horn" version, which has both a procedural and a declarative semantics. This Horn subset corresponds to the workflow control specifications of Defn 2.1. It is this property that allows a user to *program* games and strategies within the logic. Unlike classical logic programming, the proof theory presented here computes new enforcible outcomes (or m-path sets) *as well as* query answers. This section proposes an SLD-style proof theory. The soundness is easy to show but the completeness still remains to be proved.

The transaction base $\mathbf{P}$ must be a workflow control specification as defined in Defn 2.1. We say that the combination of a transaction base $\mathbf{P}$ and a generalized Horn data oracle $\mathcal{O}^d(\mathbf{D})$ is *concurrent Horn* if $\mathcal{P}$ is a set of concurrent Horn rules satisfying the following *independence condition*: `For every database state` $\mathbf{D}$`, predicate symbols occurring in rule-heads in` $\mathbf{P}$ `do not occur in rule-bodies in` $\mathcal{O}^d(\mathbf{D})$. Intuitively, the independence condition means that the database does not define predicates in terms of transactions. Thus, the rule $a \leftarrow b$ *cannot* be in the database if the rule $b \leftarrow c$ is in the transaction base.

The independence condition arises naturally in a wide variety of situations, especially (a) when the data oracle returns a set of atomic formulas, or (b) when a conceptual distinction is desired between updating actions and non-updating queries. In the former case, the database is trivially independent of $\mathbf{P}$, since an atom is a Horn rule with an empty premise. In the latter case, the logic would have two sorts of predicates, query predicates and action predicates. Action predicates would be defined only in the transaction base, and query predicates would be defined only in the database. Action predicates could be defined in terms of query predicates (e.g., to express pre-conditions and post-conditions), but not vice-versa.

4.1 Inference System

First we need the notion of the *hot component* of a formula; it is a generalization of a similar notion from [7]. Namely, $hot(\phi)$ is a set of subformulas of ϕ, defined by induction on the structure of ϕ as follows:

1. $hot(())=\{\}$, if $()$ is an atomic formula
2. $hot(\phi)=\{\phi\}$, if ϕ is an atomic formula
3. $hot(\phi \otimes \psi) = hot(\phi)$
4. $hot(\phi \mid \psi) = hot(\phi) \cup hot(\psi)$
5. $hot(\phi \vee \psi) = \{\phi \vee \psi\}$
6. $hot(\phi \sqcap \psi) = \{\phi \sqcap \psi\}$.

Note that in cases of $\vee$ and $\sqcap$, the hot component is a singleton set that contains the very formula that is used as an argument to hot. Here are some examples of hot components:

$a \otimes b \otimes c$	$\{a\}$
$(a \otimes b) \mid (c \otimes d)$	$\{a, c\}$
$(a \sqcap b) \otimes c$	$\{a \sqcap b\}$
$(a \sqcap b) \mid (c \vee d)$	$\{a \sqcap b,\ c \vee d\}$
$((a \sqcap b) \otimes c) \vee ((f \mid g) \otimes h)$	$\{((a \sqcap b) \otimes c) \vee ((f \mid g) \otimes h)\}$

Note that a hot component represents a *particular occurrence* of a subformula in a bigger formula. For instance, $hot(a \otimes b \otimes a)$ is $\{a\}$, where a corresponds to the *first* occurrence of this subformula in $a \otimes a \otimes b$ and not the second one. This point is important because in the inference rules, below, we will sometime say that ψ' is obtained from ψ by deleting a hot occurrence of a (or some other subformula). Thus, in the above, deleting the hot component a leaves us with $b \otimes a$, *not* $a \otimes b$.

The inference rules are as follows:

Axioms: $\mathbf{P}, \mathbf{D}$ --- $\vdash$ $()$, for any $\mathbf{D}$.

Inference Rules: In Rules 1–4 below, σ is a substitution, ψ and ψ' are concurrent serial conjunctions, and a is a formula in $hot(\psi)$.

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

$$\frac{\mathbf{P}, \mathbf{D} \text{ ---} \vdash (\exists)\, \psi'\, \sigma}{\mathbf{P}, \mathbf{D} \text{ ---} \vdash (\exists)\, \psi} \quad \text{where } \psi' \text{ is } \psi \text{ where a hot occurrence of } a \text{ is replaced by } \beta.$$

   For instance, if $\psi = (c \sqcap e) \mid (a \otimes f) \mid (d \vee h)$ and the hot component in question is a in the middle subformula, then $\psi' = (c \sqcap e) \mid (\beta \otimes f) \mid (d \vee h)$.

2. *Querying the database:* If $\mathcal{O}^d(\mathbf{D}) \models^c (\exists) a\sigma$; $a\sigma$ and $\psi'\sigma$ share no variables; then

$$\frac{\mathbf{P}, \mathbf{D} \text{ ---} \vdash (\exists)\, \psi'\, \sigma}{\mathbf{P}, \mathbf{D} \text{ ---} \vdash (\exists)\, \psi} \quad \text{where } \psi' \text{ is obtained from } \psi \text{ by deleting a hot occurrence of } a.$$

   For instance, if $\psi = (c \sqcap e) \mid (a \otimes f) \mid (d \vee h)$ and the hot component is a in the middle subformula, then $\psi' = (c \sqcap e) \mid f \mid (d \vee h)$.

3. *Executing elementary updates:* If $\mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2) \models^c (\exists) a\sigma$; $a\sigma$ and $\psi'\sigma$ share no variables, then

$$\frac{\mathbf{P}, \mathbf{D}_2 \text{ ---} \vdash (\exists)\, \psi'\, \sigma}{\mathbf{P}, \mathbf{D}_1 \text{ ---} \vdash (\exists)\, \psi} \quad \text{where } \psi' \text{ is obtained from } \psi \text{ by deleting a hot occurrence of } a.$$

   For instance, if $\psi = (c \sqcap e) \mid (a \otimes f) \mid (d \vee h)$ and the hot component is a in the middle subformula, then $\psi' = (c \sqcap e) \mid f \mid (d \vee h)$.

4. *Player's move:* Let ψ be a formula with a hot component, η, of the form $\alpha \vee \beta$. Then we have the following two inference rules, which might lead to two *independent* possible derivations.

$$\frac{\mathbf{P}, \mathbf{D} \text{ ---} \vdash (\exists)\, \psi'}{\mathbf{P}, \mathbf{D} \text{ ---} \vdash (\exists)\, \psi} \qquad \frac{\mathbf{P}, \mathbf{D} \text{ ---} \vdash (\exists)\, \psi''}{\mathbf{P}, \mathbf{D} \text{ ---} \vdash (\exists)\, \psi}$$

   Where ψ' is obtained from ψ by replacing the hot component η with α and ψ'' is obtained from ψ by replacing η with β.

   For instance, if $\psi = (c \sqcap e) \mid (a \otimes f) \mid (d \vee h)$ and the hot component η is $d \vee h$, then $\psi' = (c \sqcap e) \mid (a \otimes f) \mid d$ and $\psi'' = (c \sqcap e) \mid (a \otimes f) \mid h$.

5. *Opponent's move:* Let ψ be a formula with the hot component, τ, of the form $\gamma \sqcap \delta$. Then we have the following inference rule:

$$\frac{\mathbf{P}, \mathbf{D} \text{ ---} \vdash (\exists)\, \psi' \quad \text{and} \quad \mathbf{P}, \mathbf{D} \text{ ---} \vdash (\exists)\, \psi''}{\mathbf{P}, \mathbf{D} \text{ ---} \vdash (\exists)\, \psi}$$

   where ψ' is obtained from ψ by replacing the hot component τ with γ and ψ'' is obtained from ψ by replacing τ with δ.

   For instance, if $\psi = (c \sqcap e) \mid (a \otimes f) \mid (d \vee h)$ and the hot component τ is $c \sqcap e$, then $\psi' = c \mid (a \otimes f) \mid (d \vee h)$ and $\psi'' = e \mid (a \otimes f) \mid (d \vee h)$.

   Note that unlike in the player's case when we have two inference rules, the opponent's case is a *single* inference rule. It says that in order to prove $\mathbf{P}, \mathbf{D}$ --- $\vdash$ $(\exists)\, \psi$ (*i.e.*, to execute ψ on a set S of paths emanating from $\mathbf{D}$) we need to be able to execute (or prove) ψ' on a set S_1 of paths emanating from $\mathbf{D}$ and ψ'' on another (possibly the same) set S_2 of paths emanating from $\mathbf{D}$ such that $S = S_1 \cup S_2$.

5 Modeling Workflow Constraints in Game-$\mathcal{CTR}$

In [13], we have shown how a large class of constraints on workflow execution can be expressed in $\mathcal{CTR}$. In Game-$\mathcal{CTR}$ we are interested in finding a similar class of constraints. As in [13], our results require that workflows have no loops in them, which is captured by the *unique event property* defined below. [5]

We assume that there is a subset of propositions, $\mathcal{EVENTS}$, 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 3.2, these propositions constitute the built-in elementary updates.

[5] This assumption has good practical reasons. It is made by virtually all formal approaches to workflow modeling (*e.g.*, [2, 24] and even a recent comprehensive proposal from IBM for a Web service language WSDL [18].

Definition 5.1 *(Unique Event Property).* A Game-$\mathcal{CTR}$ workflow $\mathcal{W}$ has the $\boldsymbol{unique\ event\ property}$ if and only if every proposition in $\mathcal{E}_{VENT}$ can execute at most once in any execution of $\mathcal{W}$. Formally, this can be defined both semantically and syntactically. The syntactic definition is that for every $e \in \mathcal{E}_{VENT}$:

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 5.2 *(Constraints).* Let ϕ be a $\mathcal{CTR}$ formula (*i.e.*, without $\sqcap$). 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 Game-$\mathcal{CTR}$, but this is a side issue and we will not do it here.

Our constraints on workflow execution, defined below, will all be of the form $*\phi$. Rules 1–3 define *primitive* constraints, denoted $\mathcal{P}_{RIMITIVE}$. Rule 4 defines the set $\mathcal{C}_{ONSTR}$ of all constraints.

1. **Elementary primitive constraints**: If $e \in \mathcal{E}_{VENT}$ 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 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 3.2). The latter constraint, $*(\neg e)$, means that e does not occur on any m-path in S in this sense.

   Both of these constraints can be expressed in $\mathcal{CTR}$ and hence in Game-$\mathcal{CTR}$, but we will not do this here since this is a side issue (see [13]).

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

3. **Serial primitive constraints**: If $e_1, ..., e_n \in \mathcal{E}_{VENT}$ 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, $\mathcal{C}_{ONSTR}$, consists of all Boolean combinations of primitive constraints using the connectives $\vee$ and $\wedge$.

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 $*(e_1 \otimes e_2) \wedge *(e_2 \otimes e_3)$. To get a better feel for the constraints in $\mathcal{C}_{ONSTR}$, we present a few typical examples 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 (before or after e).
This is known as Klein's existence constraint [17].

Example 5.3 *(Procurement Workflow).* The following is the specification of a procurement workflow among two agents, *buyer* and *seller,* and two services, *financing* and *delivery.*

$$buyer \leftarrow pay\_escrow \otimes (financing \mid seller)$$
$$financing \leftarrow (approve \otimes (make\_payment \vee cancel)) \sqcap (reject \otimes cancel)$$
$$seller \leftarrow reserve\_item \otimes (delivery \vee recv\_escrow)$$
$$delivery \leftarrow insured \vee non\_insured$$
$$insured \leftarrow (delivered \otimes pay\_back) \sqcap (lost \otimes pay\_back)$$
$$non\_insured \leftarrow (delivered \otimes pay\_back) \sqcap lost$$

Note that the financing service depends on actions of a client that are beyond the control of the scheduler. The client can choose to go through or cancel. Likewise, the insurance service depends on what happens during the delivery

— also beyond the scheduler's control. The contract among the above actors can be expressed using the following constraints:

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

6 Synthesizing a Coordinator by Solving Workflow Games

Given a Game-$\mathcal{CTR}$ workflow $\mathcal{W}$ and a "winning condition" for the scheduler specified as a constraint, $\Phi \in \mathcal{C}_{ONSTR}$, we would like to construct a workflow $\mathcal{W}_\Phi$ that represents a class of "winning strategies," *i.e.*, strategies that guarantee that Φ can be *enforced* by the scheduler. "Enforcement" here means that as long as `opponent` plays by the workflow rules (*i.e.*, can only choose the alternatives specified by the $\sqcap$-connective), the scheduler can ensure that every play satisfies the constraints. In Game-$\mathcal{CTR}$ this amounts to computing $\mathcal{W} \wedge \Phi$. Intuitively $\mathcal{W} \wedge \Phi$ represents the collection of all `player`'s strategies for finding the outcomes of $\mathcal{W}$ that satisfy the constraint Φ. Note that if $\mathcal{W}$ has the unique event property (Definition 5.1), then so does $\mathcal{W} \wedge \Phi$.

We must clarify what we mean by "computing" $\mathcal{W} \wedge \Phi$ and why. In the (Game-$\mathcal{CTR}$) logical sense, $\mathcal{W} \wedge \Phi$ already *is* a representation of all the winning strategies for the scheduler and all outcomes of $\mathcal{W} \wedge \Phi$ satisfy Φ. However, this formula cannot be viewed as an explicit set of instructions to the workflow scheduler. In fact, according to Definition 2.1, this formula is *not* a workflow at all. On the other hand, the formula $\mathcal{W}_\Phi$ that we are after *is* a workflow. It does not contain the $\wedge$-connective, and it can be directly executed by the Game-$\mathcal{CTR}$ proof theory.

6.1 Algorithm for Generating Strategies

As we just explained, given a Game-$\mathcal{CTR}$ workflow $\mathcal{W}$ and a winning condition Φ, we need 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 Game-$\mathcal{CTR}$ proof theory and this can be thought of as a set of executable strategies that the workflow coordinator can follow to enforce Φ.

Enforcing complex constraints. Let $*C_1, *C_2 \in \mathcal{C}_{ONSTR}$, $\mathcal{W}$ be a Game-$\mathcal{CTR}$ workflow, then

$$\begin{aligned}(*C_1 \vee *C_2) \wedge \mathcal{W} &\equiv (*C_1 \wedge \mathcal{W}) \vee (*C_2 \wedge \mathcal{W}) \\ (*C_1 \wedge *C_2) \wedge \mathcal{W} &\equiv (*C_1 \wedge (*C_2 \wedge \mathcal{W}))\end{aligned}$$

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 Game-$\mathcal{CTR}$ unique-event workflow) and returns a Game-$\mathcal{CTR}$ workflow all of whose outcomes satisfy the constraint. Let $\alpha, \beta \in \mathcal{E}_{VENT}$ and $\mathcal{W}_1, \mathcal{W}_2$ be Game-$\mathcal{CTR}$ workflows. Then:

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

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

The above transformations are identical to those used for workflows of cooperating tasks in [13]. The first transformation below is specific to Game-$\mathcal{CTR}$. Here we use σ to denote $*\alpha$ or $*\neg\alpha$:

$$\begin{aligned}\sigma \wedge (\mathcal{W}_1 \sqcap \mathcal{W}_2) &= (\sigma \wedge \mathcal{W}_1) \sqcap (\sigma \wedge \mathcal{W}_2) \\ \sigma \wedge (\mathcal{W}_1 \vee \mathcal{W}_2) &= (\sigma \wedge \mathcal{W}_1) \vee (\sigma \wedge \mathcal{W}_2)\end{aligned}$$

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

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

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

Let $\alpha, \beta \in \mathcal{E}_{\mathcal{VENT}}$ and let $\mathcal{W}$ be a Game-$\mathcal{CTR}$ workflow. Then:

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

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: $\mathtt{sync}(\alpha < \beta, \mathcal{W}) \ = \ \mathcal{W}'$, 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 the facts of the form *channel(...)* are assumed to be different from all the other facts that might appear in the database states that the workflows travel through during the execution. Thus, $send(\xi)$ inserts some new constant into the communication channel and $receive(\xi)$ deletes it. 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 6.1 (Enforcing Elementary and Serial Constraints). *The above transformations for elementary and serial primitive constraints are correct, i.e., they compute a Game-$\mathcal{CTR}$ 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 6.2 (Disjunctive Primitive Constraints). *Let σ_i be elementary constraints. Then*

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

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 (considered as Game-$\mathcal{CTR}$ formulas). These transformations are correct only for workflows that satisfy the unique-event property.

Definition 6.3 *(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. $(\mathcal{W} \ \wedge \ (\varphi \mid \mathtt{Playset})) \rightarrow *\sigma$
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 6.4 *(Co-occurrence 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. $(\mathcal{W} \ \wedge \ (\varphi \mid \mathtt{Playset})) \rightarrow (\psi \mid \mathtt{Playset})$,
2. ϕ and ψ are disjoint subformulas in $\mathcal{W}$, and

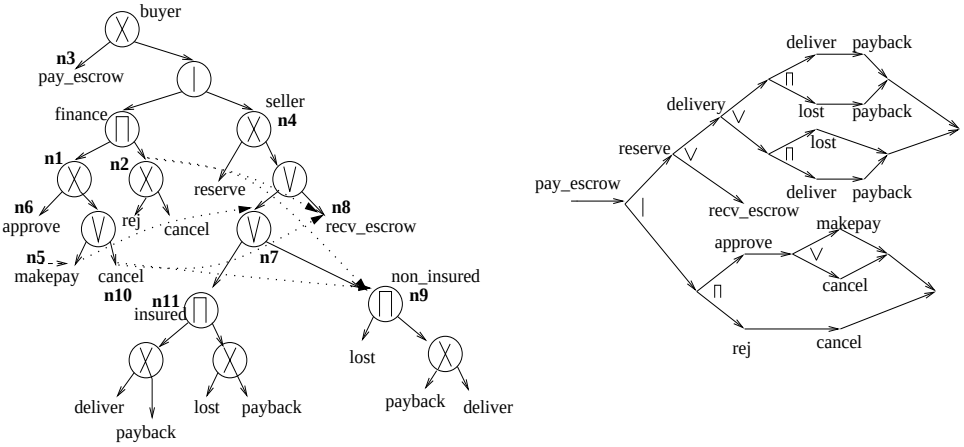

Figure 1: Workflow parse tree and workflow graph for Example 5.3

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 6.5 (Computation of $GS_{*\sigma}(\mathcal{W})$ 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 6.6 *(Computation of maximal guarantees and coexecution).* The workflow parse tree for Example 5.3 is shown in Figure 1(a). 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 coexecuting 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 $n2$.

The algorithm in Figure 2 computes a workflow that is equivalent to $((*\sigma_1 \wedge ... \wedge *\sigma_n \rightarrow *\sigma) \wedge \mathcal{W})$ (*i.e.*, it 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 4-5, we identify the subformulas h of $\mathcal{W}$ that coexecute 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 5.1), 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 6) *until* it is guaranteed

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 **then**
6. rewrite f to $send(\xi) \otimes f$ and set $Guard(g) := Guard(g) \cup \{recv(\xi)\}$, for all $g \in GS_{*\neg\sigma}(h)$
7. **else** $solve((*\sigma_1 \wedge ... \wedge *\sigma_n \rightarrow *\sigma),\ sibling(f))$
8. **for** each $g \in subformulas(\mathcal{W})$ **do**
9. **if** $Guard(g) \neq \emptyset$ **then** rewrite g to $\big(\bigvee_{recv(\xi) \in Guard(g)} recv(\xi)\big) \otimes g$

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

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 7, 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 8-9, 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 6.7 *(Procurement Workflow (contd.)).* The algorithm in Figure 2 creates the following workflow by applying the constraints in Example 5.3 to the procurement workflow in that example. The parse tree for that workflow is depicted in Figure 1(a).

- 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(a) as a dotted line. This ensures that whenever the buyer cancels the procurement workflow, the seller collects the escrow.

- To enforce $(*approve\ \wedge\ *\neg cancel \rightarrow *pay\_back)$, 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 $GS_{*\neg pay\_back}(n4) = \{n8, n9\}$, 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 $GS_{*\neg payback}(n4) = \{n8, n9\}$, 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 the seller can no longer pocket the escrow and that delivery must use the insured method. The resulting strategy is:

  $buyer \leftarrow pay\_escrow \otimes (financing \mid seller)$
  $financing \leftarrow (approve \otimes ((send(\xi_1) \otimes make\_payment) \vee ((send(\xi_3) \otimes cancel)))$
  $\qquad\qquad \sqcap (send(\xi_2) \otimes (reject \otimes cancel))$
  $seller \leftarrow reserve\_item \otimes ((recv(\xi_1) \otimes delivery) \vee ((recv(\xi_2) \vee recv(\xi_3)) \otimes recv\_escrow))$
  $delivery \leftarrow insured \vee ((recv(\xi_2) \vee recv(\xi_3)) \otimes non\_insured)$
  $insured \leftarrow (delivered \otimes pay\_back) \sqcap (lost \otimes pay\_back)$
  $non\_insured \leftarrow (delivered \otimes pay\_back) \sqcap lost$

Proposition 6.8 (Enforcing disjunctive primitive constraints). *The above algorithm for enforcing disjunctive primitive constraints is correct, i.e., it computes a Game-$\mathcal{CTR}$ workflow that is equivalent to $\mathcal{W} \wedge *(\sigma_1 \vee ... \vee \sigma_n)$ where σ_i are* elementary constraints.

Theorem 6.9 (Complexity for enforcing disjunctive primitive constraints). *Let $\mathcal{W}$ be a control flow graph and $*\Phi \in \mathcal{P}_{\mathcal{RIMITIVE}}$ be a disjunctive primitive constraint. 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 6.10 (Complexity of solving games). *Let $\mathcal{W}$ be a control flow graph $\mathcal{W}$ and $\Phi \subset \mathcal{C}_{\mathcal{ONSTR}}$ be a set of global constraints in the conjunctive normal form $\bigwedge_N(\vee_j Prim)$ where $Prim \in \mathcal{P}_{\mathcal{RIMITIVE}}$. 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. After compiling the constraints Φ into the workflow $\mathcal{W}$, several things still need to be done. The first 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*.

Example 6.11 *(Cyclic Block).* 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 `player` moves into b, a deadlock occurs. However, we can rewrite this workflow into $a \otimes (c \sqcap d)$ to avoid the problem.

Definition 6.12 *(Workflow control flow graph).* Let $\mathcal{W}$ be a workflow control specification. Then the corresponding ***workflow control flow graph***, G(W), is defined as follows: For *atomic* α, we introduce two nodes, begin(α) and end(α), and add an arc from begin(α) to end(α), labeled with α. For $\phi \otimes \psi$ add an edge from end(ϕ) to begin(ψ). The node end($\phi \otimes \psi$) is defined to coincide with end(ψ) and begin($\phi \otimes \psi$) with begin(ϕ). For $\phi \ \mathbf{op} \ \psi$, where $\mathbf{op} \in \{\vee, |, \sqcap\}$, indroduce new nodes, begin($\phi \ \mathbf{op} \ \psi$) and end($\phi \ \mathbf{op} \ \psi$). Add the arcs from begin($\phi \ \mathbf{op} \ \psi$) to begin($\phi$) and to begin($\psi$). Add directed edges from end(ϕ) and end(ψ) to end($\phi \ \mathbf{op} \ \psi$). The workflow graph for Example 5.3 is illustrated in Figure 1(b).

The problem with *cyclic blocks* is that, when they exist, finding an execution path in $\mathcal{W}$ may not be a linear task and the proof procedure of Game-$\mathcal{CTR}$ would have to explore multiple paths simultaneously (because some paths may get stuck in the 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 $O(|\mathcal{W}|^3)$. This procedure, which we call `Unblock`, is shown in Figure 3. It either yields a cyclic-block-free Game-$\mathcal{CTR}$ workflow that is equivalent to $\mathcal{W}$ or $\neg$`Playset`, if $\mathcal{W}$ is inconsistent. In the figure, $\mathcal{G}$ denotes the control flow graph for $\mathcal{W}$.

procedure `Unblock`($\mathcal{G}$)
1. **for** each $cycle_i \in \mathcal{G}$ **do** using DFS
2. **for each** $recv(\xi_{ij}) \in cycle_i$ **do**
3. $\mathcal{W}_{ij} = remove(\mathcal{W}, recv(\xi_{ij}))$
4. return $\bigvee \mathcal{W}_{ij}$

procedure $remove(\mathcal{W}, recv(\xi))$
1. $\mathcal{W}' := (\mathcal{W} \wedge \neg recv(\xi))$
2. $\mathcal{S} := S(\mathcal{W}) - S(\mathcal{W}')$ s.t. $S(\mathcal{W}) = \{send(\xi) \mid send(\xi) \in \mathcal{W}\}$
3. **if** $\mathcal{S} \neq \emptyset$ **then**
4. **for each** $send(\xi_i) \in \mathcal{S}$ **do**
5. $\mathcal{W}' := remove(\mathcal{W}', recv(\xi_i))$
6. **else return** $\mathcal{W}'$

Figure 3: An Algorithm for Removing Cyclic Blocks

Simplification. Even after all these steps the result can still have literals of the form $\neg$`Playset` so, strictly speaking, $\mathcal{W}_\Phi$ is not a Game-$\mathcal{CTR}$ workflow. These literals can occur either due to enforcement of elementary constraints (see the beginning of Section 6.1) or due to the elimination of cyclic blocks. These literals can be removed with the help of the following Game-$\mathcal{CTR}$ tautologies:

$$\neg\texttt{Playset} \otimes \varphi \ \equiv \ \varphi \otimes \neg\texttt{Playset} \ \equiv \ \neg\texttt{Playset}$$
$$\neg\texttt{Playset} \mid \varphi \ \equiv \ \varphi \mid \neg\texttt{Playset} \ \equiv \ \neg\texttt{Playset}$$
$$\neg\texttt{Playset} \vee \varphi \ \equiv \ \varphi \vee \neg\texttt{Playset} \ \equiv \ \varphi$$
$$\neg\texttt{Playset} \sqcap \varphi \ \equiv \ \varphi \sqcap \neg\texttt{Playset} \ \equiv \ \neg\texttt{Playset}$$

The result would be either a Game-$\mathcal{CTR}$ workflow or $\neg$`Playset`.

7 Conclusion and Related Work

Process algebras and alternating temporal logic [10, 1] have been used for modeling open systems with game semantics, where system and the environment interacts. Model checking is a standard mechanism for verifying temporal properties of such systems and deriving automata for scheduling. In [19], 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. However, the winning conditions in [19] are very different form ours in nature.

The use of Game-$\mathcal{CTR}$ has enabled us to find a more efficient verification algorithm than what one would get using model checking. Indeed, standard model checking techniques [10, 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 6.10), 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.

In addition, Game-$\mathcal{CTR}$ is interesting on its own. Logic games have been proposed before in other contexts [16, 22]. As in Game-$\mathcal{CTR}$, validity of a statement in such a logic means that the `player` has a winning strategy against the `opponent`. In Game-$\mathcal{CTR}$ however, games, winning conditions, and strategies are *themselves* logical formulas (rather than modal operators). Logical equivalence in Game-$\mathcal{CTR}$ is a basis for constructive algorithms for solving games and synthesizing strategies, which yield strategies that are executable by the proof theory of Game-$\mathcal{CTR}$. Related game logic formalisms, such as [16, 22], 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 workflows, but *not* for *programming* and *scheduling* them.

Related work in planning, where planning goals are expressed as temporal formulas includes [4]. In [4], 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 model checking approaches.

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. Attie, M. Singh, A. Sheth, and M. Rusinkiewicz. Specifying and enforcing intertask dependencies. In *Intl. Conference on Very Large Data Bases*, Dublin, Ireland, August 1993.

[3] P.C. Attie, M.P. Singh, E.A. Emerson, A.P. Sheth, and M. Rusinkiewicz. Scheduling workflows by enforcing intertask dependencies. volume 3, pages 222–238, December 1996.

[4] 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.

[5] A.J. Bonner and M. Kifer. Transaction logic programming. In *Intl. Conference on Logic Programming*, pages 257–282, Budapest, Hungary, June 1993. MIT Press.

[6] A.J. Bonner and M. Kifer. An overview of transaction logic. *Theoretical Computer Science*, 133:205–265, October 1994.

[7] 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.

[8] 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.

[9] Y. Breitbart, A. Deacon, H.J. Schek, A. Sheth, and G. Weikum. Merging application-centric and data-centric approaches to support transaction-oriented multi-system workflows. *SIGMOD Record*, 22(3), 1993.

[10] 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.

[11] F. Curbera, W.A. Nagy, and S. Weerawarana. Web services: Why and how. In *OOPSLA 2001 Workshop on Object-Oriented Web Services*. ACM, 2001.

[12] H. Davulcu, M. Kifer, C.R. Ramakrishnan, and I.V. Ramakrishnan. Logic based modelling and analysis of workflows. in preparation, October 1997.

[13] 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.

[14] Eds. A. Dogac, L. Kalinichenko, M.T. Ozsu, and A. Sheth. *Workflow Management Systems and Interoperability*. Springer, NATO ASI Series. Computer and Systems Science, Vol 164, 1998.

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

[16] J. Hintikka. *Logic, Language Games, and Information*. Clarendon, Oxford, 1973.

[17] J. Klein. Advanced rule-driven transaction management. In *IEEE COMPCON*. IEEE, 1991.

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

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

[20] M.E. Orlowska, J. Rajapakse, and A.H.M. ter Hofstede. Verification problems in conceptual workflow specifications. In *Intl. Conference on Conceptual Modelling*, volume 1157 of *Lecture Notes in Computer Science*, Cottbus, Germany, 1996. Springer-Verlag.

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

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

[23] T.S. Siegfried. Semantic web service architecture — evolving web service standards toward the semantic web. URL: http://citeseer.nj.nec.com/461405.html.

[24] 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.

[25] 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.