Logic Based Modeling and Analysis of Workflows *
(Extended Abstract)

Hasan Davulcu
SUNY at Stony Brook
davulcu@cs.sunysb.edu

Michael Kifer
SUNY at Stony Brook
kifer@cs.sunysb.edu

C.R. Ramakrishnan
SUNY at Stony Brook
cram@cs.sunysb.edu

I.V. Ramakrishnan
SUNY at Stony Brook
ram@cs.sunysb.edu

Abstract

We propose Concurrent Transaction Logic ($\mathcal{CTR}$) as the language for specifying, analyzing, and scheduling of workflows. We show that both local and global properties of workflows can be naturally represented as $\mathcal{CTR}$ formulas and reasoning can be done with the use of the proof theory and the semantics of this logic. We describe a transformation that leads to an efficient algorithm for scheduling workflows in the presence of global temporal constraints, which leads to decision procedures for dealing with several safety related properties such as whether every valid execution of the workflow satisfies a particular property or whether a workflow execution is consistent with some given global constraints on the ordering of events in a workflow. We also provide tight complexity results on the running times of these algorithms.

1 Introduction

A *workflow* is a collection of cooperating, coordinated activities designed to carry out a well-defined complex process, such as trip planning, graduate student registration procedure, or a business process in a large enterprise. An activity in a workflow might be performed by a human, a device, or a program. *Workflow management systems* provide a framework for capturing the interaction among the activities in a workflow and are recognized as a new paradigm for integrating disparate systems, including legacy systems [20, 8]. Ideally, they should also help the user in analysis and reasoning about complex business processes.

It has been realized that analysis and reasoning about workflows requires a formal specification model with a well defined semantics [18, 2]. In this paper, we develop a novel framework for *specifying*, *analyzing* and *executing* workflows based on Transaction Logic [5, 4, 6, 7].

Workflow representation frameworks. Figure 1 depicts three most common frameworks for specifying workflows: *control flow graph*, *triggers* (also known as *event-condition-action rules*), and *temporal constraints*.

The control flow graph is most appropriate for depicting the local execution dependencies of the activities in a workflow; it is a good way to visualize the overall flow of control. Control flow graphs are the primary specification means in most commercial implementations of workflow management systems. A typical graph specifies the initial and the final activity in a workflow, the successor-activities for each activity in the graph, and whether these successors must *all* be executed concurrently, or it suffices to execute just one branch non-deterministically. In Figure 1, all successors of activity **a** must be executed, which is indicated with the "AND"-label. In contrast, "OR" indicates that when **b** is finished, there is a choice of executing **d**, **h**, then **j** *or* **e** then **j**. Successful execution of any one of these branches should suffice for the overall success of the workflow.

Arcs in a control flow graph can be labeled with *transition conditions.* The condition applies to the current state of the workflow (which, in a broad sense, may include the current state of the underlying database, the output of the completed tasks, the current time, etc.). When the task at the tail of an arc completes, the task at the head can begin only if the corresponding transition condition evaluates to true.

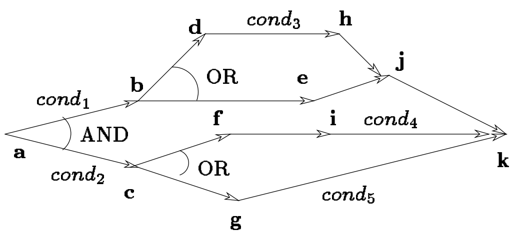
The Workflow Management Coalition [10] identifies additional controls, such as loops and sub-workflows. Various researchers have also suggested other types of controls, including alternative execution and compensation for failed activities [12, 17, 15, 25, 28, 1, 13]. However, control flow graphs have one obvious limitation: they cannot be used to specify *global* dependencies between workflow tasks, such as those expressed as global constraints on the right-hand side of Figure 1.

Defining workflows using triggers is yet another possibility [11]. However, this method is not as general as control flow graphs. For instance, like the graphs, triggers cannot be used to specify global task dependencies, and they are not sufficiently expressive when it comes to representing alternatives in workflow execution (depicted as "OR" nodes in Figure 1). In fact, it follows from a result in [7] that triggers with so-called "immediate" execution semantics can be represented using control flow graphs, and this result can be adapted to triggers with the "eventual" execution semantics as well. Since triggers can be "compiled into" the control flow graph, we shall be treating triggers as part of the control flow graph.

Other researchers proposed frameworks that rely exclusively on constraints to specify *both* local and global prop-

*This work is partially supported by a DLA/DARPA contract and by the NSF grants IRI-9404629, CCR-9705998, 9711386, 9510072 9404921, CDA-9504275, 9303181, INT-9600598

Control Flow Graph



Triggers

If event **d** occurs **then**
If $cond_6$ holds **then**
execute activity **p**

If event **g** occurs **then**
If $cond_7$ holds **then**
execute activity **q**

Global Constraints

If **d** is ever executed **then**
g must also be executed

If **e** and **i** are both executed
then **e** must happen before **i**

If **g** and **j** are both executed
then **j** must happen before **g**

Figure 1: Frameworks for Specifying Workflows

erties of workflows. In [26, 27], Singh describes an algebra of temporal constraints, which is believed to cover all useful global dependencies that might arise in workflow systems. For instance, this algebra includes Klein's constraints [22], which commonly occur in workflow specifications [1]. This algebra is sufficiently expressive for modeling control flow graphs that have *no* transition conditions attached to arcs. However, such constraints cannot be used to model workflows that query the intermediate state of the workflow and make scheduling decisions based on the outcome. More importantly, algebraic approaches are not part of a larger reasoning framework and so it is unclear how they apply to more expressive specification languages (*e.g.*, to represent sub-workflows, failure and compensation, etc.) and how they could be used for reasoning and verifying the correctness of workflows.

Logic-based formalism. In this paper, we base our approach on Concurrent Transaction Logic [6] (abbr., $\mathcal{CTR}$). There are many reasons for this choice. First, control flow graphs with transition conditions can be easily and naturally represented in $\mathcal{CTR}$. Second, [7] shows that triggers are easy to represent as well. Finally, [5] contains an extensive discussion of the temporal capabilities of $\mathcal{CTR}$. In particular, the entire algebra of constraints described in [26] is isomorphic to a small subset of the propositional Transaction Logic. Hence $\mathcal{CTR}$ provides us with a unifying formalism that subsumes all of the three popular workflow specification frameworks described above. Furthermore, being a full-blown logic, $\mathcal{CTR}$ allows not only scheduling workflows, but also *reasoning* about their properties. Finally, like in logic programming systems, the proof theory of $\mathcal{CTR}$ is also a run-time environment for *executing* workflows.

Summary of results. Our approach is based on a transformation procedure, named `Apply`, which accepts a workflow specification that includes a control flow graph $\mathcal{G}$, triggers (viewed as part of $\mathcal{G}$), and a set of temporal constraints $\mathcal{C}$, and constructs an equivalent specification in $\mathcal{CTR}$ that represents only those executions of $\mathcal{G}$ where $\mathcal{C}$ holds. The resulting specification can be directly used to execute the workflow. Thus, `Apply` can be viewed as a *compilation* process that facilitates verification of workflows and optimizes run-time scheduling.

[1] Klein constraints are of the form: (1) if events **a** and **b** both occur, then **a** occurs earlier than **b**; or (2) if event **a** ever occurs then **b** must occur as well (before or after **a**).

1. `Apply` transformation enables us to determine:
   (a) Whether every legal execution of a given workflow specification satisfies a particular property. Moreover, if some execution does not satisfy the property, then the verification procedure returns a counter example which is the most general execution of the workflow that violates the property in question.
   (b) Whether the specification made up of the control flow graph and global constraints is consistent; and
   (c) Whether some of the specified constraints are redundant.
2. The transformation eliminates the parts of the control graph that are inconsistent with the constraints, which facilitates scheduling of events at run-time.
3. The separation of control flow graph and global constraints in the workflow specifications leads to tighter complexity results for the verification problem.

Our results also contribute to the theory of Transaction Logic itself. Here, we essentially extend the efficient SLD-style proof procedure of $\mathcal{CTR}$ from so called concurrent-Horn goals to a larger class of formulas, which incorporates temporal constraints (a more precise formulation appears in Section 2).

2 An Overview of Concurrent Transaction Logic

This section provides a short summary of the $\mathcal{CTR}$ syntax, which is used in this paper to represent workflows. Due to space limitation, we cannot discuss the model theory of the logic or its proof theory. Instead, we rely on the procedural reading of $\mathcal{CTR}$ statements.[2] A thorough treatment of the main aspects of Transaction Logic appears in [6, 5, 4]. A fairly detailed, yet informal introduction can be found in [21].

$\mathcal{CTR}$ is a conservative extension of the classical predicate logic in the sense that both its proof theory and the model theory reduce to those of the classical logic for formulas that do not cause state transitions (but only query the current state).

[2] This is analogous to the procedural reading of Datalog programs.

Basic syntax. The atomic formulas of $\mathcal{CTR}$ are identical to those of the classical logic, *i.e.*, they are expressions of the form $p(t_1, \ldots, t_n)$, where p is a predicate symbol and the t_i's are function terms. More complex formulas are built with the help of connectives and quantifiers.

Apart from the classical $\vee$, $\wedge$, $\neg$, $\forall$, and $\exists$, $\mathcal{CTR}$ has two additional connectives, $\otimes$ (*serial conjunction*) and $\mid$ (*concurrent conjunction*), and two modal operators, $\diamond$ (*executional possibility*) and $\odot$ (*isolated execution*). For instance, $\odot(p(X) \otimes q(X)) \mid (\forall Y(r(Y) \vee s(X,Y)))$ is a well-formed formula.

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

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

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

With this in mind, the intended meaning of the $\mathcal{CTR}$ connectives can be summarized as follows:

- $\phi \otimes \psi$ *means*: execute ϕ then execute ψ. Or, model-theoretically, $\phi \otimes \psi$ is true over a path $\langle s_1, \ldots, s_n \rangle$ if ϕ is true over a prefix of that path (say, $\langle s_1, \ldots, s_i \rangle$) and ψ is true over the suffix (*i.e.*, $\langle s_i, \ldots, s_n \rangle$). In terms of control flow graphs (cf. Figure 1), this connective represents arcs connecting adjacent activities.

- $\phi \mid \psi$ *means*: ϕ and ψ must both execute concurrently, in an interleaved fashion. This connective corresponds to the "AND"-nodes in control flow graphs.

- $\phi \wedge \psi$ *means*: ϕ and ψ must both execute along the *same* path. In practical terms, this is best understood in terms of *constraints* on the execution. For instance, ϕ can be thought of as a transaction and ψ as a constraint on the execution of ϕ. It is this feature of the logic that lets us specify temporal constraints as part of workflow specifications.

- $\phi \vee \psi$ *means*: execute ϕ *or* execute ψ non-deterministically. This connective corresponds to the "OR"-nodes in control flow graphs.

- $\neg\phi$ *means*: execute in any way, provided that this will not be a valid execution of ϕ. There are many uses for this feature. One is that, just as in classical logic, the negation lets us define deductive rules which, in terms of the workflows, correspond to sub-workflow definitions. Negation is also an important component in temporal constraint specifications.

- $\odot\phi$ *means*: execute ϕ in isolation, *i.e.*, without interleaving with other concurrently running activities.

  This operator enables us to specify the transactional parts of workflow specifications.

- $\diamond\phi$ *means*: check if ϕ is executable at the current state. Section 7 discusses the role of the possibility operator $\diamond$ in workflow modeling.

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

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

Having provided the intuition behind the logical connectives, it is now easy to see how control flow graphs are represented in $\mathcal{CTR}$. For instance, the graph in Figure 1 is represented as:

$$a \otimes \left(\left(cond_1 \otimes b \otimes ((d \otimes cond_3 \otimes h) \vee e) \otimes j \right) \mid \left(cond_2 \otimes c \otimes ((f \otimes i \otimes cond_4) \vee (g \otimes cond_5)) \right) \right) \otimes k \tag{1}$$

Expressions of the above form are called *concurrent-Horn goals*. Formally:

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

It should be clear from the above example how control flow graphs translate into concurrent-Horn goals.

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

In this paper, we limit our attention to non-iterative workflows, which means that we do not allow recursive concurrent rules. Section 7 discusses to what extent our present results apply to recursively defined workflows.

From the workflow point of view, the primary use for the rules is to represent sub-workflows. Indeed, since workflows and sub-workflows can be described using concurrent-Horn goals, we can use the rules of the form *subWorkFlowName* $\leftarrow$ *subWorkFlowDefinition* to define sub-workflows. For instance, *subWorkFlowName* can be used in workflow specifications as if it were a regular activity, thereby completely hiding the underlying structure of the activity from top-level specifications.

Observe that the definition of concurrent-Horn rules and goals *does not* include the connective $\wedge$. In general, $\wedge$ represents *constrained execution*, which is usually hard to implement, since constraints must be checked at every step of the execution. If a constraint violation is detected, a new execution path must be tried out. In contrast, the concurrent-Horn fragment of $\mathcal{CTR}$ is efficiently implementable, and there exist an SLD-style proof procedure that proves concurrent-Horn formulas and *executes* them at the same time [6].

The efficiency gap between concurrent-Horn execution and constrained execution is the main motivation for our results. In logical terms, we show that, for a large class of constraints, formulas of the form *ConcurrentHornGoal* $\wedge$ *Constraints* have an equivalent concurrent-Horn form (which, therefore, does not use the connective $\wedge$). In practical terms, therefore, this means that there is an efficient workflow scheduling strategy and, moreover, this strategy can be determined at "design time" of the workflow (as opposed to run-time scheduling of [27]).

Elementary updates. We complete our informal introduction to $\mathcal{CTR}$ by explaining how execution of (some) formulas may actually change the underlying database state. Most of the machinery has already been introduced (albeit very informally). What is missing is the notion of *elementary updates.*

In $\mathcal{CTR}$, elementary updates are represented by ordinary atomic, variable-free formulas. Syntactically, $\mathcal{CTR}$ does not distinguish elementary updates in any way, but the user may want to do so by adopting a syntactic convention (*e.g.*, a convention could be that $ins.p(t)$ represents the act of insertion of tuple t into the relation p).

What distinguishes elementary updates is their semantics. Through some black magic, called *transition oracle*, $\mathcal{CTR}$ arranges so that each elementary updates is always true along certain arcs, *i.e.*, paths of the form $\langle s_1, s_2\rangle$. Informally, one can think of an elementary update as a binary relation over states. For instance, if $\langle s_1, s_2\rangle$ belongs to the relation corresponding to an elementary update u, it means that u can cause a transition from state s_1 to state s_2. Note that an update can be *non-deterministic* (any one of a number of alternative state transitions might be possible) and it is possible for an update to be inapplicable in certain states (but it is also possible for an update to apply in every state).[3]

This mechanism is very general. It accounts for a wide variety of elementary state changes: from simple tuple insertions and deletions, to relational assignments, to updates performed by legacy programs, to whatever workflow activities might do. The connectives of $\mathcal{CTR}$ are then used to build more complex updates from the elementary ones and then to combine these complex updates into even more complex update programs. This process of building $\mathcal{CTR}$ programs from the ground up is very natural and powerful. The reader is referred to [4, 5, 6] for concrete examples.

Now we can explain how the various workflow activities (*e.g.*, the symbols a, b, c, etc., in (1)) appear to $\mathcal{CTR}$. Namely, each activity is encoded as a variable-free atomic formula, η, that represents either a sub-workflow defined by a set of concurrent-Horn rules, or it can represent an ordinary activities, in which case η is an elementary update. The latter is appropriate, since individual activities appear to workflow management systems as "black boxes" that perform state changes in ways that are (at best) only partially specified.

3 Events and Temporal Constraints

In workflow systems, tasks are typically modeled in terms of their significant, externally observable events, such as *start*, *commit*, or *abort*. For the purpose of control flow, we can represent these events as regular activities and incorporate them directly into the control flow graph in appropriate spots. The temporal constraints on workflow execution can then be expressed in terms of these events. Without loss of generality (as far as workflow modeling goes), we make the following assumptions:

- *No significant event occurs twice during the execution.* Indeed, we can always rename different occurrences of the same type of event.
- *Each significant event is represented as an elementary* update that applies in every state.
  This assumption is appropriate since, typically, a significant event amounts to nothing more than forcing a suitable record into the system log.

(2)

The first assumption translates into the following *unique event property*, which limits the kinds of concurrent-Horn goals that we shall consider in this paper:

Definition 3.1 ***(Unique Event Property).*** A concurrent-Horn goal $\mathcal{G}$ has the *unique event property* if and only if every significant event occurs at most once in any execution of $\mathcal{G}$. In such cases, we shall also say that $\mathcal{G}$ is a *unique-event goal.* □

Unique-event goals can be recognized in linear time in the size of the goal, but we shall not present this algorithm here. Instead, we mention some obvious, yet useful properties of such goals, which suffices for our purposes. Let α be a significant event. Then:

- If $G = E_1 \otimes E_2$ is a unique-event goal and α occurs in E_1 then it cannot occur in E_2.
- If $G = E_1 \mid E_2$ is a unique-event goal and α occurs in E_1 then it cannot occur in E_2.
- If $G = E_1 \vee E_2$ then G is a unique-event goal if and only if so are both E_1 and E_2.

(3)

In the rest of this paper, all concurrent-Horn goals are assumed to have the unique event property.

Transaction Logic can express a wide variety of temporal constraints [5], but here we focus on a relatively simple algebra of constraints, which we denote by $\mathcal{C}\mathrm{ONSTR}$. $\mathcal{C}\mathrm{ONSTR}$ is as expressive as Singh's Event Algebra [27]. Using these constraints we can specify that one task must start before some other task, that the execution of one task causes some other task to be executed or not executed, etc. These constraints are believed to be sufficient for the needs of workflow management systems, and they are far beyond of the capabilities of the currently available commercial systems.

We specify all significant events in the system as propositions drawn from a set, denoted by $\mathcal{E}\mathrm{VENT}$. In addition, we introduce one special proposition, $\mathtt{path}$, which is defined as $\phi \vee \neg\phi$, for any $\mathcal{CTR}$ formula. This means that $\mathtt{path}$ is true on *all* possible execution paths.[4]

Definition 3.2 ***(Constraints).*** The basic building blocks of $\mathcal{C}\mathrm{ONSTR}$ are formulas of the form $\mathtt{path} \otimes e \otimes \mathtt{path}$, where $e \in \mathcal{E}\mathrm{VENT}$. To save space, we shall use a shortcut for such formulas: $\nabla\phi \;\equiv\; \mathtt{path} \otimes \phi \otimes \mathtt{path}$, by definition. Then the following constraints form the constraint algebra $\mathcal{C}\mathrm{ONSTR}$:

[3] An example of the first kind is an update that deletes $p(t)$ only if $p(t)$ is true in the current state. An example of the second update is deletion of $p(t)$ regardless of whether $p(t)$ is true. If $p(t)$ is not true in some state, s, then no state transition takes place, but the update will still be true over the arc $\langle s, s\rangle$.

[4] This is one of the counterparts of "true" in classical logic. In $\mathcal{CTR}$, one can define other propositions that express various truths. For instance, we can express the proposition $\mathtt{state}$, which is true precisely on paths of length 1, *i.e.*, at states. It is also possible to express formulas that are true precisely on arcs, etc.

1. **Primitive constraints**: If $e \in \mathcal{E}\mathcal{VENT}$ then ∇e (event e must happen) and $\neg\nabla e$ (event e must *not* happen) are *primitive* constraints in $\mathcal{C}\mathcal{ONSTR}$. The constraint ∇e is a *positive* primitive constraint and $\neg\nabla e$ is a *negative* primitive constraint.

2. **Serial constraints**: If $s_1, ..., s_n \in \mathcal{C}\mathcal{ONSTR}$ are *positive* primitive constraints, then $s_1 \otimes \cdots \otimes s_n \in \mathcal{C}\mathcal{ONSTR}$ is a *serial* constraint. For convenience, primitive constraints are also viewed as serial constraints.

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

Nothing else is in $\mathcal{C}\mathcal{ONSTR}$. □

To get a better grasp of the capabilities of $\mathcal{C}\mathcal{ONSTR}$, here are a few typical constraints and their real-world interpretation:

- $\nabla e \wedge \nabla f$ — events e and f must both occur (in some order);
- $\neg\nabla e \vee \neg\nabla f$ — it is not possible for e and f to happen together.
- $\neg\nabla e \vee (\nabla e \otimes \nabla f)$ — if event e occurs, then f must occur some time later;
- $\neg\nabla e \vee \neg\nabla f \vee (\nabla e \otimes \nabla f)$ — if both e and f occur, then e must come before f. This is known as Klein's order constraint [22].
- $\neg\nabla f \vee (\nabla e \otimes \nabla f)$ — if event f has occurred, then event e must have occurred some time prior to that;
- $\neg\nabla e \vee \nabla f$ — if event e occurs, then f must also occur (before or after e). This is known as Klein's existence constraint [22].

Note that Definition 3.2 does not explicitly state that $\mathcal{C}\mathcal{ONSTR}$ is closed under negation. Nevertheless, we can show that it is.

Proposition 3.3 (Splitting Serial Constraints). *Under the assumptions (2), any serial constraint is equivalent to a $\wedge$-conjunction of serial constraints, each composed of no more than two primitive constraints.*

Proof. Consider a positive serial constraint composed of more than two primitive constraints: $\nabla e_1 \otimes \nabla e_2 \otimes s$, where s is a serial constraint. We can show that this is equivalent to $(\nabla e_1 \otimes \nabla e_2) \wedge (\nabla e_2 \otimes s)$. □

A serial constraint of the form $\nabla\alpha \otimes \nabla\beta$ is called an *order constraint*; it says that α and β must both occur and α must occur before β. (Note that this is somewhat stronger than Klein's order constraint mentioned earlier.)

Lemma 3.4 (Constraint Negation). *Let $C \in \mathcal{C}\mathcal{ONSTR}$. Then $\mathcal{C}\mathcal{ONSTR}$ has a constraint that is equivalent to $\neg C$ under the assumptions (2).*

Proof. We can push negation down to the serial constraints in C using the classical De Morgan's laws for $\wedge$, $\vee$, and $\neg$, which are valid also in $\mathcal{CTR}$:

$$\begin{array}{lcl} \neg(\phi \vee \psi) & \equiv & \neg\phi \wedge \neg\psi \\ \neg(\phi \wedge \psi) & \equiv & \neg\phi \vee \neg\psi \\ \neg\neg\phi & \equiv & \phi \end{array}$$

Since $\neg\neg\nabla e$ is equivalent to ∇e, we only need to show that $s = \neg(\nabla e_1 \otimes \cdots \otimes \nabla e_n)$ is equivalent to some constraint in $\mathcal{C}\mathcal{ONSTR}$.

By Proposition 3.3, we can assume that $n < 3$. If $n = 1$, then $s = \neg\nabla e_1$ is a negative primitive constraint. If $n = 2$, then $s = \neg(\nabla e_1 \otimes \nabla e_2)$, which is equivalent (under the assumptions (2)) to $\neg\nabla e_1 \vee \neg\nabla e_2 \vee (\nabla e_2 \otimes \nabla e_1)$. □

The previous results lead to the following normal form for the members of $\mathcal{C}\mathcal{ONSTR}$:

Corollary 3.5 (Normal Form for Constraints). *Every constraint in $\mathcal{C}\mathcal{ONSTR}$ is equivalent to a constraint of the form $\vee_i(\wedge_j \mathrm{serialConstr_{i,j}})$ where each $\mathrm{serialConstr_{i,j}}$ is either a primitive constraint or a serial constraint composed of two positive primitive constraints.*

Proof. Follows from Proposition 3.3, Lemma 3.4, and the fact that, as in classical logic, $\vee$ distributes through $\wedge$ and vice versa. □

Lemma 3.4 helps express certain constraints much more easily. For instance:

- $\neg(\nabla e \otimes \nabla f)$ — it is not possible for f to occur after e (and for e before f).
- $\neg(\nabla e \otimes \nabla f \otimes \nabla g)$ — if e happens and then f does, the event g cannot happen later.

4 Consistency, Verification, and Scheduling Problems for Workflows

This section and the next assumes that the control flow graphs do not have transition conditions on the arcs and that the specification does not contain concurrent-Horn rules that define sub-workflows. In Section 7, we discuss how our results apply to graphs that include these features.

Let $\mathcal{G}$ be a concurrent-Horn goal with unique event property (Definition 3.1), which represents a control flow graph, and let $\mathcal{C} \subset \mathcal{C}\mathcal{ONSTR}$ be a set of constraints. The three central problems in workflow management systems can be formulated as follows:

Consistency: *Determine whether $\mathcal{G}$ is consistent with $\mathcal{C}$.*

Verification: *Determine whether every legal execution of the workflow satisfies some property $\Phi \in \mathcal{C}\mathcal{ONSTR}$.*

Scheduling: *Find an execution path (or all paths) in $\mathcal{G}$ where $\mathcal{C}$ holds.*

In $\mathcal{CTR}$, the consistency problem is tantamount to the existence of an execution for the formula $\mathcal{G} \wedge \mathcal{C}$.

The verification problem is a special case of the consistency problem. Indeed, every legal execution of the workflow satisfies Φ iff $\mathcal{G} \wedge \mathcal{C} \wedge \neg\Phi$ cannot execute (*i.e.*, $\mathcal{G}$ is inconsistent with $\mathcal{C} \wedge \neg\Phi$).

The verification problem also subsumes the *redundancy* problem: $\Phi \in \mathcal{C}$ is redundant iff every legal execution of $\mathcal{G} \wedge (\mathcal{C} - \{\Phi\})$ satisfies Φ.

In this paper, we solve the verification problem constructively by transforming the formula $\mathcal{G} \wedge \mathcal{C}$ into an *equivalent concurrent-Horn* formula $\mathcal{G}'$, which is always executable; or if this is impossible, $\mathcal{G} \wedge \mathcal{C}$ reduces to $\neg\mathtt{path}$ — a non-executable transaction, which is the $\mathcal{CTR}$ analog of the classical *false*. Our algorithm is exponential in the size of $\mathcal{C}$ (in the worst case), which turns out to be inherent to the verification problem:

Proposition 4.1 (Complexity of Verification). *Let $\mathcal{G}$ be a concurrent goal and $\mathcal{C} \subset \mathcal{C}onstr$ be a set of constraints. Then determining whether $\mathcal{G} \wedge \mathcal{C}$ is executable in $\mathcal{CTR}$ is NP-complete.*

The NP-hardness proof is by reduction to satisfiability of propositional logic [16]. That the decision problem is in NP follows from the fact that given an arbitrary sequence of events the satisfiability of a set of constraints and a unique-event control flow graph is decidable in polynomial time. A similar result has been previously obtained in [24]. However, their NP-completeness result is based on *synchronizer*-constraints. Each synchronizer corresponds to a combination of an *existence constraint*[5] and an *order constraint*[6] in our formalism. We tighten their complexity result by showing that synchronization per se is not a culprit: the problem is NP-complete even in the presence of just the existence constraints. In fact, it follows from our solution to the consistency problem that for order constraints the verification problem can be solved in polynomial time.

The scheduling problem needs more explanation. Workflow literature distinguishes two approaches to the problem: *passive* and *pro-active*.

Passive schedulers receive sequences of events from an external source, such as a workflow or a transaction manager, and validate that these sequences satisfy all global constraints (possibly after reordering some events in the sequences). Several such schedulers are described in [26, 3, 19]. To validate a particular sequence of events, each of these schedulers takes at least quadratic time in the number of events.[7] However, in passive scheduling environments, it is left to an unspecified external system to do consistency checking, to ensure the liveness of the scheduling strategy and to select the event sequences for validation. The known algorithms for these tasks are worst-case exponential.

In contrast to passive scheduling, our approach is pro-active. In particular, we do not rely on any external system. Instead, we construct a "compressed" explicit representation of all allowed executions (*i.e.*, executions that are known to satisfy all constraints). This representation can be used to enumerate all allowed executions at linear time per execution path (linear in the size of the path). In this way, at each stage in the execution of a workflow, the scheduler knows all events that are eligible to start. There is no need to validate constraints at run time, since the constraints are "compiled into" the structure.

More precisely, our solution to the scheduling problem capitalizes on the solution to the consistency problem. First, we verify that the specifications are consistent by transforming $\mathcal{G} \wedge \mathcal{C}$ into an equivalent concurrent-Horn goal $\mathcal{G}'$, as explained above. The formula $\mathcal{G}'$ plays the role of the aforesaid explicit representation for the set of all allowed executions of $\mathcal{G} \wedge \mathcal{C}$.

If the transformation succeeds (*i.e.*, the specifications are consistent), enumerating all execution paths of $\mathcal{G}'$ takes time linear in $\mathcal{G}$ per path (note: linear in the original graph, not in the much larger graph $\mathcal{G}'$!). This means that after the compilation, we can pick a legal schedule for workflow activities in time *linear* in the size of the original control flow graph. In contrast, the event scheduler of [27] has quadratic complexity.

Thus, while expanding the effort on consistency checking (which needs to be done anyway), we compile the original specifications into a form that lets us find allowable schedules much more efficiently than with the passive approaches of [27, 3, 19] (It should be noted that, these latter algorithms do not do consistency checks).

[5] Existence constraints form the subset of $\mathcal{C}onstr$ obtained from primitive constraints by combining them with $\wedge$ and $\vee$ only.

[6] Order constraints form the subset of $\mathcal{C}onstr$ that does not use $\vee$.

[7] Two of these schedulers are actually exponential in the size of the largest global constraint, but it is reasonable to assume that in practice this size is bound by a small constant.

5 Compiling Constraints into the Control Flow Graph

We define the process of compiling the constraints in $\mathcal{C}onstr$ into unique-event goals by starting with simple events and extending the transformation to more complex ones. The unique-event property assumption is crucial for the correctness of the results in this section.

Compiling primitive constraints.

The following transformation takes a primitive constraint of the form $\nabla\alpha$ or $\neg\nabla\alpha$ and a control flow graph (expressed as a concurrent unique-event goal) and returns a concurrent-Horn goal whose executions are precisely those executions of the original graph that satisfy the constraint. Intuitively, this means that the contraint is compiled into the graph.

Definition 5.1 ***(Primitive Constraint Compilation).*** Let $\alpha, \beta \in \mathcal{E}vent$. Then:

$$\begin{array}{lll} \mathtt{Apply}\ (\nabla\alpha,\ \alpha) & = & \alpha \\ \mathtt{Apply}\ (\nabla\alpha,\ \beta) & = & \neg\mathtt{path} \qquad \text{if } \alpha \neq \beta \\ \mathtt{Apply}\ (\neg\nabla\alpha,\ \alpha) & = & \neg\mathtt{path} \\ \mathtt{Apply}\ (\neg\nabla\alpha,\ \beta) & = & \beta \qquad \text{if } \alpha \neq \beta \end{array}$$

Let T and K be concurrent-Horn goals and let σ stand for $\nabla\alpha$ or $\neg\nabla\alpha$. Then:

$$\begin{array}{ll} \mathtt{Apply}(\nabla\alpha,\ T \otimes K) & = \begin{array}{l}(\mathtt{Apply}(\nabla\alpha,\ T) \otimes K) \vee \\ (T \otimes \mathtt{Apply}\ (\nabla\alpha,\ K))\end{array} \\ \mathtt{Apply}(\neg\nabla\alpha, T \otimes K) & = \mathtt{Apply}(\neg\nabla\alpha, T) \otimes \mathtt{Apply}(\neg\nabla\alpha, K) \\ \mathtt{Apply}(\nabla\alpha,\ T \mid K) & = \begin{array}{l}(\mathtt{Apply}(\nabla\alpha,\ T) \mid K) \vee \\ (T \mid \mathtt{Apply}\ (\nabla\alpha,\ K))\end{array} \\ \mathtt{Apply}(\neg\nabla\alpha, T \mid K) & = \mathtt{Apply}(\neg\nabla\alpha, T) \mid \mathtt{Apply}(\neg\nabla\alpha, K) \\ \mathtt{Apply}(\sigma,\ \odot T) & = \odot(\mathtt{Apply}(\sigma,\ T)) \\ \mathtt{Apply}(\sigma,\ T \vee K) & = \mathtt{Apply}(\sigma,\ T) \vee \mathtt{Apply}\ (\sigma,\ K) \qquad \Box \end{array}$$

Observe that, due to the properties given in (3), the above transformation preserves the unique-event property of concurrent-Horn goals. For example, if T is $\gamma \otimes (\alpha \vee \beta \vee \eta) \otimes \delta$, then:

$$\begin{array}{lll} \mathtt{Apply}(\nabla\alpha,\ T) & = & \gamma \otimes \alpha \otimes \delta \\ \mathtt{Apply}(\neg\nabla\alpha,\ T) & = & \gamma \otimes (\beta \vee \eta) \otimes \delta \end{array}$$

Proposition 5.2 (Primitive Constraint Compilation). *If T is a concurrent-Horn goal and σ is a primitive constraint, then* $\mathtt{Apply}(\sigma,\ T) \ \equiv\ T \wedge \sigma$.

Compiling order constraints.

Next we extend `Apply` to work with order constraints, *i.e.*, constraints of the form $\nabla\alpha \otimes \nabla\beta$.

Definition 5.3 ***(Order Compilation).*** Let $\alpha, \beta \in \mathcal{E}vent$ and let T be a concurrent-Horn goal. Then:

$$\begin{array}{l} \mathtt{Apply}(\nabla\alpha \otimes \nabla\beta, T) = \\ \mathtt{sync}(\alpha < \beta, \mathtt{Apply}(\nabla\alpha,\ \mathtt{Apply}(\nabla\beta,\ T))) \end{array}$$

The transformation sync is designed to synchronize events in the desired order. It is defined as follows:

$$\mathtt{sync}(\alpha < \beta, T) \;=\; T'$$

where T' is like T, except that every occurrence of event α is replaced with $\alpha \otimes send(\xi)$ and every occurrence of event β is replaced with $receive(\xi) \otimes \beta$, where ξ is a new constant.

The actions *send* and *receive* are easily expressed in $\mathcal{CTR}$ (see [6]) and their semantics is what one would expect of such synchronization primitives: $receive(\xi)$ is true if and only if $send(\xi)$ has been previously executed. In this way, β cannot start before α is done. □

It is easy to verify that, due to (3), the above transformation preserves the unique-event property of concurrent-Horn goals. The following examples illustrate the transformation:

$$\begin{array}{l} \mathtt{Apply}(\nabla\alpha \otimes \nabla\beta,\ \gamma \vee (\beta \otimes \alpha)) = \\ receive(\xi) \otimes \beta \otimes \alpha \otimes send(\xi) \\ \mathtt{Apply}(\nabla\alpha \otimes \nabla\beta,\ \alpha \mid \beta \mid \rho_1 \mid \ldots \mid \rho_n) \;= \\ (\alpha \otimes send(\xi)) \mid (receive(\xi) \otimes \beta) \mid \rho_1 \mid \ldots \mid \rho_n \end{array} \qquad (4)$$

Proposition 5.4 (Order Compilation). *Let T be a concurrent-Horn goal and $\alpha, \beta \in \mathcal{E}vent$. Then* $\mathtt{Apply}(\nabla\alpha \otimes \nabla\beta,\ T) \;\equiv\; T \wedge (\nabla\alpha \otimes \nabla\beta)$.

Compiling general constraints.

We are now ready to extend Apply to handle the general constraints in $\mathcal{C}onstr$.

*Definition 5.5 **(Compiling General Constraints)**.* Let T be a concurrent-Horn goal. We assume that workflows are specified by a set of constraints $\mathcal{C}$ and each individual constraint is represented in the normal form of Corollary 3.5. Therefore, $\mathcal{C}$ can be written as a single dependency of the form

$$\delta_1 \wedge \delta_2 \wedge \ldots \wedge \delta_n \qquad (5)$$

where each δ_i is in the normal form. In particular, all serial constraints are assumed to have been split into simpler order constraints. To extend Apply to such constraints, we only need to define:

$$\begin{array}{ll} \mathtt{Apply}(C_1 \vee C_2, T) & \equiv \mathtt{Apply}(C_1,\ T) \;\vee\; \mathtt{Apply}(C_2,\ T) \\ \mathtt{Apply}(C_1 \wedge C_2, T) & \equiv \mathtt{Apply}(C_1,\ \mathtt{Apply}(C_2,\ T)) \end{array} \quad □$$

As before, it is easy to see that the above transformation preserves the unique-event property.

Proposition 5.6 (Compiling General Constraints). *Let T be a concurrent-Horn goal and let δ be a constraint of the form (5). Then* $\mathtt{Apply}(\delta,\ T) \;\equiv\; T \wedge \delta$.

Knots.

After compiling the constraints $\mathcal{C}$ into the graph $\mathcal{G}$, several things still need to be done. First, the result of the compilation, $\mathcal{G}_{\mathcal{C}}$, can have literals of the form $\neg\mathtt{path}$ so, strictly speaking, $\mathcal{G}_{\mathcal{C}}$ is not a concurrent-Horn goal. However, we can use the following $\mathcal{CTR}$ tautologies to simplify $\mathcal{G}_{\mathcal{C}}$:

$$\begin{array}{l} \neg\mathtt{path} \otimes \phi \;\equiv\; \phi \otimes \neg\mathtt{path} \;\equiv\; \neg\mathtt{path} \\ \neg\mathtt{path} \mid \phi \;\equiv\; \phi \mid \neg\mathtt{path} \;\equiv\; \neg\mathtt{path} \\ \neg\mathtt{path} \vee \phi \;\equiv\; \phi \vee \neg\mathtt{path} \;\equiv\; \phi \end{array}$$

The result would be either a concurrent-Horn goal or $\neg\mathtt{path}$.

If the result is not $\neg\mathtt{path}$, this still does not mean that we have a directly executable workflow specification. The reason is that the *send/receive* synchronization primitives may cause a "deadlock". In model-theoretic terms, this means that such a formula is $\mathcal{CTR}$-equivalent to $\neg\mathtt{path}$, and in proof-theoretic terms this means that the proof procedure would halt and declare that no execution exists. In this case, we rewrite $\mathcal{G}_{\mathcal{C}}$ into $\neg\mathtt{path}$.

Also, when the proof procedure declares a failure, it produces a concurrent Horn goal, $\mathcal{G}_{fail}$, which in a sense is the smallest subpart of the original workflow that is inconsistent with the constraints. In this way, the workflow designers can be given a feedback that might help them find the bug in their specifications.

Even if the proof procedure of $\mathcal{CTR}$ does find a proof and thus $\mathcal{G}_{\mathcal{C}}$ is an executable workflow specification, $\mathcal{G}_{\mathcal{C}}$ may have sub-formulas where the *send/receive* primitives cause a cyclic wait, which we call *knots*.

The problem with knots is that, when they exist, finding an execution path in $\mathcal{G}_{\mathcal{C}}$ may not be a linear task (despite what we have promised in Section 4). Fortunately, it is easy to show that a variant of the proof theory of $\mathcal{CTR}$ can be used to remove all knots from $\mathcal{G}_{\mathcal{C}}$ in time linear in the size of $\mathcal{G}_{\mathcal{C}}$. This procedure, which we call Excise, yields either a knot-free concurrent-Horn goal equivalent to $\mathcal{G}_{\mathcal{C}}$, or $\neg\mathtt{path}$, if $\mathcal{G}_{\mathcal{C}}$ is inconsistent.

We illustrate the Excise process with the following example.

*Example 5.7 **(Knots)**.* Let the graph $\mathcal{G}$ be $\gamma \otimes (\eta \vee (\alpha \mid \beta \mid \eta))$ and let the constraints be as follows: $c_1 \equiv \neg\nabla\alpha \vee (\nabla\alpha \otimes \nabla\beta)$, $c_2 \equiv \neg\nabla\beta \vee (\nabla\beta \otimes \nabla\eta)$, $c_3 \equiv \neg\nabla\alpha \vee (\nabla\eta \otimes \nabla\alpha)$. The constraint c_1 says, *If α takes place, then β must also happen afterwards.* The other constraints have similar interpretation. Omitting some intermediate steps, we have:

$$\begin{array}{l} \mathtt{Apply}(c_1, \mathcal{G}) = \mathtt{Apply}(\neg\nabla\alpha, \mathcal{G}) \vee \mathtt{Apply}(\nabla\alpha \otimes \nabla\beta, \mathcal{G}) \\ \qquad = \mathcal{G}_1 \vee \mathcal{G}_2, \\ \qquad \text{where } \mathcal{G}_1 \equiv \gamma \otimes (\alpha \otimes send(\xi_1) \mid receive(\xi_1) \otimes \beta \mid \eta) \\ \qquad \text{and } \mathcal{G}_2 \equiv \gamma \otimes \eta \\ \mathtt{Apply}(c_2, \mathcal{G}_1 \vee \mathcal{G}_2) = \mathtt{Apply}(c_2, \mathcal{G}_1) \vee \mathtt{Apply}(c_2, \mathcal{G}_2) \\ \qquad = \mathcal{G}_3 \vee \mathcal{G}_2, \\ \qquad \text{where } \mathcal{G}_3 \equiv \gamma \otimes \big(\alpha \otimes send(\xi_1) \,\big|\, receive(\xi_1) \otimes \beta \\ \qquad \otimes send(\xi_2) \,\big|\, receive(\xi_2) \otimes \eta\big) \\ \mathtt{Apply}(c_3, \mathcal{G}_3 \vee \mathcal{G}_2) = \mathcal{G}_4 \vee \mathcal{G}_2, \\ \qquad \text{where } \mathcal{G}_4 \equiv \gamma \otimes \big(receive(\xi_3) \otimes \alpha \otimes send(\xi_1) \,\big| \\ \qquad receive(\xi_1) \otimes \beta \otimes send(\xi_2) \,\big|\, receive(\xi_2) \otimes \eta \\ \qquad \otimes send(\xi_3)\big) \end{array}$$

Finally, $\mathtt{Excise}(\mathcal{G}_4 \vee \mathcal{G}_2) = \mathtt{Excise}(\mathcal{G}_4) \vee \mathtt{Excise}(\mathcal{G}_2)$. The proof procedure of $\mathcal{CTR}$ finds no knots in $\mathcal{G}_2$, so $\mathtt{Excise}(\mathcal{G}_2) = \mathcal{G}_2$. On the other hand, it detects a knot in $\mathcal{G}_4$ as follows.

First, the proof procedure "executes" γ and deletes it from $\mathcal{G}_4$. This results in a goal where each concurrent conjunct starts with a *receive* and the corresponding *send*'s are slated to occur only later in the execution. Therefore, the proof procedure halts and we declare a knot in $\mathcal{G}_4$. Thus $\mathtt{Excise}(\mathcal{G}_4) = \neg\mathtt{path}$. Hence, $\mathtt{Excise}(\mathtt{Apply}(c_1 \wedge c_2 \wedge c_3, \mathcal{G})) \equiv \mathcal{G}_2$. □

Main results.

We are now ready to summarize how the Apply and Excise transformations help solve the consistency, verification, and related problems. Theorems 5.8 through 5.10 assume that every activity in the workflow (except for the *receive* primitive) always succeeds. Without this assumption, only the "if"-part of Theorem 5.8 holds and its corollaries, Theorems 5.9 and 5.10, must be adjusted accordingly.

Theorem 5.8 (Consistency Checking). *Given a workflow specification $\mathcal{G} \wedge \mathcal{C}$, it is inconsistent iff* $\texttt{Excise}(\texttt{Apply}(\mathcal{C}, \mathcal{G})) = \neg\texttt{path}$.

Proof. Follows from Proposition 5.6 and the soundness and completeness of $\mathcal{CTR}$ proof theory. □

Theorem 5.9 (Property Verification). *Given a workflow specification $\mathcal{G} \wedge \mathcal{C}$ and a property $\Phi \in \mathcal{C}_{\mathcal{ONSTR}}$, there is a constructive way of verifying whether every execution of the workflow satisfies Φ.*

Proof. Φ is satisfied by every execution of the workflow if and only if $\texttt{Excise}(\texttt{Apply}(\neg\Phi \wedge \mathcal{C}, \mathcal{G})) = \neg\texttt{path}$. Otherwise, $\texttt{Excise}(\texttt{Apply}(\neg\Phi \wedge \mathcal{C}, \mathcal{G}))$ rewrites to the most general counter example where Φ fails to hold. □

Theorem 5.10 (Redundancy Elimination). *Given a workflow specification $\mathcal{G} \wedge \mathcal{C}$ and a constraint $\Phi \in \mathcal{C}$, we can verify whether Φ is redundant.*

Theorem 5.11 (Complexity). *Let $\mathcal{G}$ be a control flow graph $\mathcal{G}$ and $\mathcal{C} \subset \mathcal{C}_{\mathcal{ONSTR}}$ be a set of global constraints in the normal form of Corollary 3.5. Let $|\mathcal{G}|$ denote the size of $\mathcal{G}$, N be the number of constraints in $\mathcal{C}$, and d be the largest number of disjuncts in a constraint in $\mathcal{C}$. Then*

- *The worst-case size of* $\texttt{Apply}(\mathcal{C}, \mathcal{G})$ *is* $O(d^N \times |\mathcal{G}|)$.
- *The worst-case time complexity of applying* $\texttt{Excise}$ *is proportional to the size of* $\texttt{Apply}(\mathcal{C}, \mathcal{G})$.

A simple corollary of Theorem 5.11 is: If $\mathcal{C}$ consists of serial constraints only, then $d = 1$ and the size of $\texttt{Apply}(\mathcal{C}, \mathcal{G})$ is proportional to $|\mathcal{G}|$.

6 Related Formalisms

Workflow verification. Process algebras / temporal logic suites have been used for modeling concurrent systems (akin to workflows) for over a decade now, and model checking is a pretty standard mechanism for verifying such systems.

However, the salient benefits of using $\mathcal{CTR}$ over process algebras and related formalisms are very tangible. First, $\mathcal{CTR}$ is one uniform formalism in which workflows can be specified, verified and scheduled. This should be contrasted with the use of the algebras and temporal logic for specifying workflows, model-checking for their verification, and automata for scheduling.

Second, the use of $\mathcal{CTR}$ has enabled us to find more efficient verification algorithms. Indeed, standard model checking techniques [9] used for verification are worst-case exponential in the size of the control flow graph. This is often referred to as the *state-explosion problem.* In contrast, Apply is *linear* in the size of the graph — it is exponential only in the size of the constraint set (Theorem 5.11), which is a much smaller object. In a sense, Apply (along with the proof theory of $\mathcal{CTR}$) can be viewed as specialized, more efficient model checker for the problem at hand.

Third, $\mathcal{CTR}$ integrates "process oriented" and "data oriented" features, which makes it easy to model processes that perform complex transformations over the database. In fact, extending our techniques to workflows that query the underlying database state is the next logical step for our work. In contrast, using process algebras to model database state is awkward and impractical.

Fourth, after verification, the proof theory of $\mathcal{CTR}$ can schedule workflows at time linear in the size of the original graph, but exponential in the size of the constraint set. In contrast, process scheduling using the standard toolkit of process algebras and temporal logic requires automata that are exponential in the size of the original graph.

Workflow modeling. We have already discussed formalisms used for passive workflow scheduling [26, 27, 3, 19]. In addition, general purpose process specification formalisms such as Petri-nets, state charts [29], temporal logic [14] or process algebras [23] can also be used for modeling workflows. However, we believe that our results show that $\mathcal{CTR}$ provides a much simpler and uniform way to both describe and reason about workflows.

7 Conclusion

We presented a logic-based formalism for *specifying*, *verifying*, and *executing* workflows. We developed an algorithm for consistency checking of workflows and for their property verification. The algorithm compiles global constraints on workflow execution into the control flow graph. In addition to solving the consistency and verification problems, this compilation technique helps optimize the run-time scheduling of workflow events.

Our work can be extended to handle additional features, such as the following:

Transition conditions. As a compilation technique, our algorithm cannot fully account for control flow graphs with transition conditions that query the database. For such graphs, our algorithm is sound but not complete for consistency and verification problems. However, if additional semantic information about the data base is available, it can be incorporated into our framework to yield more accurate results.

Sub-workflows. It is straightforward to augment our compilation technique to handle sub-workflows defined via concurrent-Horn rules. Furthermore, when global dependencies do not span sub-workflow boundaries, the complexity reported in Theorem 5.11 can be reduced. Indeed, it can be shown that, if M is the largest number of dependencies in a sub-workflow, then the size of $\texttt{Apply}(\mathcal{C}, \mathcal{G})$ is $O(d^M \times |\mathcal{G}|)$.

Loops and iteration. Loops in control flow graph can be expressed using recursive $\mathcal{CTR}$ rules. Our techniques assumes the unique-event property for workflow graphs. Hence this property has to be relaxed to handle workflows with loops.

Failure semantics. Failure atomicity is built into $\mathcal{CTR}$ semantics. However, more complex workflows require more advanced failure semantics, such as *compensation* [15]. Some such semantics can be expressed using the possibility operator of $\mathcal{CTR}$, $\diamond$. Work is in progress on extending our framework to handle other failure semantics.

Acknowledgements. The authors would like to thank Tony Bonner for the helpful comments on a draft of this paper.

References

[1] G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. Günthör, and C. Mohan. Advanced transaction models in workflow contexts. In *International Conference on Data Engineering*, New Orleans, Louisiana, February 1996.

[2] G Alonso, D. Agrawal, A. El Abbadi, and C. Mohan. Functionality and limitations of current workflow management systems. In *IEEE-Expert (to appear in a special issue on Cooperative Information Systems)*, 1997.

[3] P. Attie, M. Singh, A. Sheth, and M. Rusinkiewicz. Specifying and enforcing intertask dependencies. In *Intl. Conference on Very Large Data Bases*, 1993.

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

[5] A.J. Bonner and M. Kifer. Transaction logic programming (or a logic of declarative and procedural knowledge). Technical Report CSRI-323, University of Toronto, November 1995. Unpublished manuscript.

[6] 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.

[7] A.J. Bonner, M. Kifer, and M. Consens. Database programming in transaction logic. In A. Ohori C. Beeri and D.E. Shasha, editors, *Proceedings of the International Workshop on Database Programming Languages*, Workshops in Computing, pages 309–337. Springer-Verlag, February 1994. Workshop held on Aug 30–Sept 1, 1993, New York City, NY.

[8] O. Bukhres and E. Kueshn, Eds. Special issue on software support for work flow management. *Distributed and Parallel Databases—An International Journal*, 3(2), April 1995.

[9] 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.

[10] Workflow Management Coalition. Terminology and glossary. Technical Report (WFMC-TC-1011), Workflow Management Coalition, Brussels, 1996.

[11] U. Dayal, M. Hsu, and R. Ladin. Organizing long-running activities with triggers and transactions. In *ACM SIGMOD Conference on Management of Data*, 1990.

[12] A. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewcz. A multi - database transaction model for interbase. In *Intl. Conference on Very Large Data Bases*, 1990.

[13] A.K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan-Kaufmann, San Mateo, CA, 1992.

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

[15] H. Garcia-Molina and K. Salem. Sagas. In *Intl. Conference on Very Large Data Bases*, pages 249–259, May 1987.

[16] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman and Company, San Francisco, CA, 1978.

[17] D. Georgakopoulos, M. Hornick, P. Krychniak, and F. Manola. Specification and management of extended transactions in a programmable transaction environment. In *International Conference on Data Engineering*, Houston, TX, February 1994.

[18] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to infrastructure for automation. *Journal on Distributed and Parallel Database Systems*, 3(2):119–153, April 1995.

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

[20] M. Hsu, Ed. Special issue on workflow systems. *Bulletin of the Technical Committee on Data Engineering (IEEE Computer Society)*, 18(1), March 1995.

[21] M. Kifer. Transaction logic for the busy workflow professional. Unpublished manuscript, August 1996.

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

[23] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[24] 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.

[25] M. Rusinkiewicz and A. Sheth. Specification and execution of transactional workflows. In W. Kim, editor, *In Modern Database Systems: The Object Model, Interoperability, and Beyond*. ACM Press, 1994.

[26] 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.

[27] 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.

[28] H. Wachter and A. Reuter. The ConTract model. In [13], chapter 7, pages 220–263. 1992.

[29] Dirk Wodtke and Gerhard Weikum. A formal foundation for distributed workflow execution based on state charts. In *Intl. Conference on Database Theory*, pages 230–246, 1997.