1

Logic Based Approaches to Workflow Modeling and Verification

Saikat Mukherjee[1] Hasan Davulcu[2] Michael Kifer[1] Pinar Senkul[3] Guizhen Yang[1]

[1] Department of Computer Science, Stony Brook University, Stony Brook, NY 11794-4400, USA
{saikat,kifer,guizyang}@cs.stonybrook.edu
[2] Department of Computer Science & Engineering, Arizona State University, Tempe, AZ 85287-5406,USA
hdavulcu@asu.edu
[3] Department of Computer Engineering, Middle East Technical University, Ankara, 06531, Turkey
karagoz@ceng.metu.edu.tr

Abstract. A *workflow* is a collection of coordinated activities designed to carry out a well-defined complex process, such as trip planning, student registration, 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* (or *WfMS*) 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. A large workflow system might involve many disparate activities that are coordinated in complex ways and are subject to many constraints. Thus, modeling such systems and ensuring that they perform according to the specifications are not an easy task. To be able to analyze the properties of workflows, the latter must be specified using a formalism with a well-defined semantics. The popular formalisms in this area are the various logics, Petri Nets [1,35], Event-Condition-Action rules [23,15], and State Charts [36]. In this paper we survey and compare a number of *logic-based* formalisms that were proposed in the literature.

* This work was sponsored in part by the NSF grants IIS-0072927 and INT-9809945.

1 Introduction

A *workflow* is a collection of coordinated activities designed to carry out a well-defined complex process, such as trip planning, student registration, or a business process in a large enterprise. Business processes are represented as sets of *tasks*, where each task carries out some well-defined activity. An activity can be as simple as reading and approving a document or it may involve a complex process of its own. An activity can be completely automated or involve manual interaction. A *workflow management system* (WfMS) is a set of tools for defining, analyzing, and managing the execution of workflows. To design a workflow, one uses a workflow modeling language (often through a graphical interface) to specify the tasks, the flow of data, the control flow, and a set of constraints on the execution. A WfMS includes an interpreter, which understands workflow specifications, can analyze them for correctness, and schedule workflow events accordingly. During the execution, a WfMS interacts with the participants in the workflow and invokes the necessary applications when required.

Prior to the advent of workflow management systems, business processes were automated in an ad-hoc manner and each system involved one of a kind solution. It was therefore hard not only to deploy a workflow system, but also to adapt it to changing business environment. A WfMS can substantially reduce the cost of business process engineering and maintenance through

- reduced operating costs — by driving down the cost per transaction
- improved productivity — by eliminating routine and repetitive tasks
- better analysis, which simplifies the job of creating *correct* workflows and leads to higher-quality designs
- improved change management — using the tools provided by a WfMS organizations can modify workflow specifications much more easily and quickly adapt to changes in the business environment
- better decision support — the tools provided by a WfMS can be used to analyze the workflow, flag inefficiencies, and verify that the workflow specification meets its goals.

The development of robust workflow management systems is one of the most important challenges in today's information systems. Collaborative design, health-care, and Web services are some examples of applications that require automated workflow management. While traditional applications have kept researchers busy, *Web services* are driving the renewed interest in this area. From the workflow point of view, a Web service is a task with a well-defined interface. A number of proposals exist to standardize the various parts of this interface (*e.g.*, UDDI [27], WSDL [26], DAML-S [3]). The promise of Web services is that the standardized interface makes it possible to combine disparate services into complex workflows and thus many issues that arise in the context of workflows are pertinent to Web services. In addition, Web services present new, unique challenges. First, since different services are

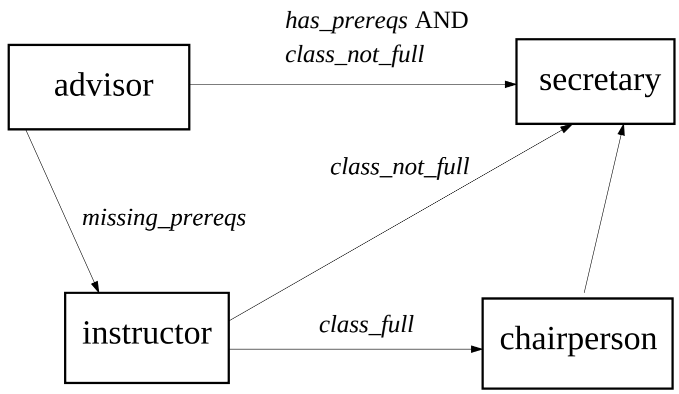


Fig. 1. Example of a registration authorization workflow

under the control of different organizations, it can no longer be assumed that they all cooperate. Therefore, it becomes necessary to be able to model workflows whose tasks might execute in adversarial environment. One work in this vein is [14]. Related to this is the potential need to negotiate services. Indeed, as consumers we do not always accept the first offer and try to find a better deal — often from the same merchant. Thus, the ability to negotiate is another unique aspect of a Web service, which brings us into the realm of *agent-based systems* [22,31].

At its core, a workflow is a process and, thus, a WfMS requires a process specification language. Figure 1 is an example, borrowed from [5], of a workflow that determines whether a student is allowed to register for a course. The process definition is comprised of different tasks, which perform the job of an advisor, instructor, etc. (as pertaining to the registration process). For instance, the task *advisor* determines whether the student has the prerequisites for the course, *instructor* may give the student permission to attend the course if the class is not full even if the student does not satisfy the prerequisites, *chairperson* can give permission to attend the course even if the class is already full, but provided that the student satisfies the prerequisites. The task *secretary* models the secretary who enters the registration information into the system. A WfMS manages the registration process by creating an instance of this process definition, which in turn contains an instance for each task. The edges connecting the activities represent the flow of control between the activities, and the labels represent transition conditions. The control flow and transition conditions are enforced by the WfMS at run time.

The problem of scheduling events that arise during the execution of a workflow can be hard both from the computational complexity point of view and also algorithmically. A business workflow can include dozens and even hundreds of tasks, which might be related through a large number of global dependencies that cannot be easily represented using a control flow graph such as the one in Figure 1. Typical global dependencies are of the form "if events A and B occur then event C must not occur" or "if A occurs then it must happen before B." Apart from scheduling, workflow specifications might need to be *verified*. One problem here is whether the constraints are consistent with each other or with other parts of the specification. If they are not, the workflow cannot be scheduled. A related problem is to find out whether a certain property is assured by the workflow specification. For instance, a mail-order workflow might need to ensure that product is not shipped prior to receiving an authorization from a credit card company. This constraint might or might not follow from the already existing constraints. Of course, we could simply add this constraint to the existing ones, but then we would have to waste time enforcing it when a verification procedure might determine that this constraint is automatically enforced if the rest of the constraints are obeyed.

The need for formal methods in workflow modeling and verification has been widely recognized [20,2] and a number of formal frameworks have been proposed. These includes event-condition-action rules (triggers) [18,23,16], the various logic-based methods [4,21,29,10,5,13,14,28], Petri Nets [35,34,1], and State Charts [36].

The past few years have seen progress in both the implementation and foundational aspects of workflow management systems. However, commercial systems offer only relatively rudimentary modeling capabilities. Specification of global inter-task constraints is still difficult and verification tools are virtually absent. These limitations force application developers to embed the enterprise logic deep into the code, which leads to considerable implementation effort and high maintenance overhead.

In this survey we discuss a number of logic-based approaches to modeling and managing workflows. In particular, we are interested in the expressive power of the approaches as well as their applicability to the problems of scheduling and verification of workflows. In Section 3 we discuss a proposal [4] based on temporal logic [17]. Section 4 presents a related approach, which is formalized using a specially devised event algebra [29]. Section 5 reviews workflow modeling techniques [13,6], whose underlying formalism is Concurrent Transaction Logic [7]. Finally, Section 7 concludes the paper and provides a brief comparison of the approaches discussed in this paper.

While we were striving for completeness, limitation of space and scope forces us to leave out some logic-based approaches, such as [5], which is based on Action Logic and triggers; ACTA [10], which attempts to formalize extended transaction models in first-order predicate logic; and Vortex [18], which uses model checking techniques to verify properties of workflows. We briefly survey these works in Section 6.

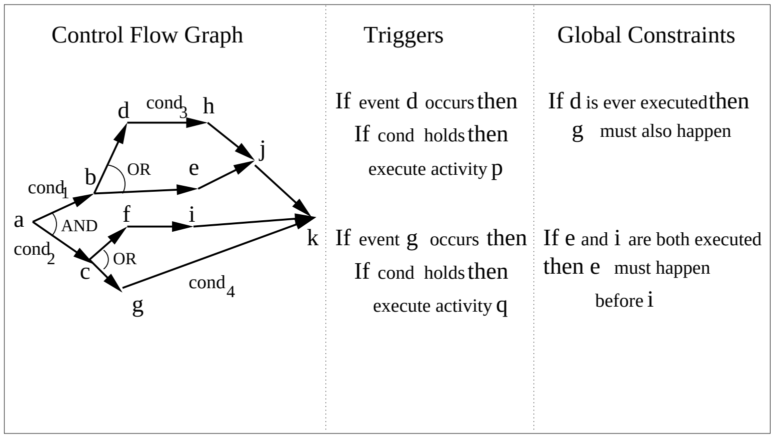


Fig. 2. Frameworks for specifying workflows

2 Preliminaries

A workflow can be modeled using one or more frameworks, and each framework can use one or more of the formalisms, such as logic, Petri Nets, etc. The most common frameworks are illustrated in Figure 2. These include *control flow graphs*, *global temporal constraints*, and *triggers*.

It should be noted that the boundary between the frameworks in Figure 2 is somewhat subjective and can vary from one approach to the next. The control flow graph can be represented as a set of constraints and some global constraints can be represented directly in the control flow graph. Likewise, some constraints can be described as triggers and some triggers can be modeled as constraints.

2.1 Modeling Concepts

The *control flow graph* is the primary technique used in commercial workflow systems. A graph specifies the initial and 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. Arcs in a control flow graph can be labeled with *transition conditions*. The condition applies to the current state of the workflow. 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 control flow graph is most appropriate for depicting the *local intertask dependencies* of the activities in a workflow. However, their limitation is that they cannot be used to specify *global intertask dependencies* between workflow tasks.[1] Dependencies (also known as *constraints*—we shall use these two terms interchangeably) provide a more general framework for specifying workflows. In particular, the precedence relationship that underlies the control flow graph is just another kind of constraint. Less obvious is the fact that both the AND-nodes and OR-nodes can be modeled as constraints (see Section 4). Nevertheless, separating global constraints from the graph is useful both as a pragmatic modeling technique and as a way to find more efficient scheduling algorithms (see Section 5). Triggers are another way of specifying control flow dependencies. Like control flow graphs, triggers are limited in their ability to specify global dependencies among tasks. They are also not sufficiently expressive when it comes to specifying alternatives in workflow execution (OR nodes).

The use of intertask dependencies for workflow modeling was first proposed in [25] and has since become a basic staple of workflow modeling works. The typical constraints on event occurrence and ordering are

- $e_1 \rightarrow e_2$ (*occurrence*): If e_1 occurs then so must be e_2. No specific ordering is implied.
- $e_1 < e_2$ (*order*): If both e_1 and e_2 occur then e_1 must be scheduled before e_2. This constraint is trivially satisfied if only one of the events occurs.

A workflow specification in the form of constraints, control flow graph, or triggers is analogous to database schema. A concrete workflow executing according to that specification is called a *workflow instance* and is analogous to database instances. Execution of a particular workflow instance is typically defined in terms of its *event history*, *i.e.*, the sequence of "significant" events (see below) that have occurred during the execution. The semantics of constraints is also defined in terms of these histories. For instance, $e_1 < e_2$ is satisfied in a given history if and only if e_1 occurs prior to e_2 in that history.

The workflow *scheduler* is a module in a WfMS that examines the incoming sequence of events, generated by the execution of workflow activities, and schedules them in a certain order so that all the given constraints would be satisfied. Alternatively, the scheduler might *pro-actively* construct a concise model of all possible executions. In this case, scheduling is essentially performed at compile time with only trivial decisions left to be made at run time. Systems that follow the former approach are described in Sections 3 and 4. An example of the latter approach appears in Section 5.

Many formal approaches to workflow modeling and scheduling (in particular, the ones surveyed in this paper) rely on some or all of the following assumptions:

[1] We shall see in Section 5 that this is theoretically possible, but practically not feasible, because compiling constraints into a control flow graph can lead to an exponential explosion of the graph. Thus, constructing such graphs manually is not an option.

Significant events: Workflow tasks are modeled as black boxes that emit *significant events*. A significant event is an abstraction that represents real events that occur during the workflow execution, which the scheduler might be interested in because they need to be put in a certain order (say, because they are mentioned in a constraint). Negation of a significant event e (denoted as $\bar{e}$ or $\neg e$) is also frequently used. The event $\bar{e}$ is said to occur if the event e never occurs in the execution.

A significant event can be one of the *standard* events, such as *start, pre-commit, commit*, and *abort*, whose semantics is known in advance, or it might be application-specific, *e.g.*, sending a message to another task. Certain constraints associated with the standard events follow from their a priori semantics. For instance, $start_T$ must precede any other event of the same task ($start_T < event_T$) and a *termination* event, $commit_T$ or $abort_T$, must be the last event in any task ($event_T < commit_T$, $event_T < abort_T$). Similarly, $commit_T$ and $abort_T$ cannot happen in the same execution of a workflow ($commit_T \rightarrow \overline{abort_T}$ and $abort_T \rightarrow \overline{commit_T}$).

For application-specific events, the workflow designer typically specifies constraints explicitly, as they depend on the application domain. For instance, the business rules of an enterprise might require that if task *shipProduct* commits then task *confirmPayment* must commit prior to that (*i.e.*, $commit_{shipProduct} \rightarrow commit_{confirmPayment}$ and $commit_{confirmPayment} < commit_{shipProduct}$).

Unique event property: No event can occur more than once in the execution of the same instance of a workflow. The rationale behind this assumption is that an event is associated with a task and a time-stamp, so it cannot occur more than once in any given execution sequence.

This assumption does not preclude events of the same *type* to occur more than once. For instance, a task can send a message to another task several times. However, if multiple events of the same type are allowed to occur, this presents a problem for the constraint specification language. For instance, how can one specify that a response to a request must follow the request? Such a statement requires that events have properties (such as event ID and context), which can be used to match a response to the corresponding request.

Forcible, rejectable, and delayable events: Some formalizations assume that significant events have certain attributes that the scheduler can use in making its decisions. These attributes are:

- *Forcible*: an event is forcible if the scheduler is permitted to make the event happen. Of course, constraints must be satisfied, but the decision whether to start such an event or not is scheduler's prerogative. For example, *abort* and *start* are forcible, since the scheduler can always abort a running task or start another task. If $start_{T_1} \rightarrow abort_{T_2}$ is a constraint, and task T_1 has already started, the scheduler might decide to force the abort of T_2 in order to satisfy the constraint.

In contrast, the scheduler might not be allowed to send messages to tasks on behalf of other tasks, so such events are not be forcible.

- *Rejectable*: an event is rejectable if the scheduler is free to prevent this event from happening. For example, the scheduler has the discretion to prevent any task from committing its work or from starting (again, subject to constraints).
  To see where this is useful, suppose the constraint is $start_{T_1} < commit_{T_2}$ and T_2 has already committed. If the event $start_{T_1}$ arrives later, the scheduler can still ensure that the constraint is satisfied by rejecting this event.
- *Delayable*: an event is delayable if the scheduler is free to delay the execution of that event. For instance, if a task has requested to commit, the scheduler might decide to delay scheduling of this event.
  Delaying is typically done to make sure that certain constraints are satisfied. For example, if $start_{T_1} < commit_{T_2}$ is a constraint and T_2 requests to commit, the scheduler might decide to delay this event until T_1 starts.

2.2 Example

In order to illustrate the different notions of events and constraints introduced in this section let us consider an airline reservation workflow. The tasks associated with this workflow are:

- *Buy*: buying an airline ticket
- *Book*: booking a car

The significant events associated with the task *Buy* are $start_{Buy}$, $commit_{Buy}$, and $abort_{Buy}$ which respectively start, commit and abort the task *Buy*. The significant events associated with the task *Book* are $start_{Book}$, $commit_{Book}$, $abort_{Book}$, and $cancel_{Book}$ which respectively start, commit, abort, and cancel the task *Book*. Observe that while booking a car can be canceled, buying an airline ticket cannot be canceled. A dependency associated with this workflow is that *Buy* commits only if *Book* commits. This can be represented as the occurrence constraint $commit_{Buy} \rightarrow commit_{Book}$. Also, if the *Buy* aborts then *Book* should also abort. This can be similarly represented as the constraint $abort_{Buy} \rightarrow abort_{Book}$. Another dependency in this workflow is that if both *Book* and *Buy* commit then *Book* commits before *Buy*. This can be represented as the order constraint $commit_{Book} < commit_{Buy}$. Yet another dependency in the workflow is that *Book* is canceled if and only if *Book* commits and *Buy* aborts. This can be modeled as the pair of occurrence constraints $cancel_{Book} \rightarrow (commit_{Book} \wedge abort_{Buy})$ and $(commit_{Book} \wedge abort_{Buy}) \rightarrow cancel_{Book}$. All the above mentioned events are delayable by the scheduler. If the event $commit_{Buy}$ happens before $commit_{Book}$ then the scheduler can delay the acceptance of $commit_{Buy}$ in order to satisfy the dependency that

Book commits before *Buy*. An example of a forcible event is $cancel_{Book}$ since the scheduler has to force that event in order to satisfy constraints when *Book* commits and *Buy* aborts. On the other hand, if *Buy* aborts and the event $commit_{Book}$ is to be scheduled, then the scheduler has to reject $commit_{Book}$ in order to satisfy the dependencies. Thus, $commit_{Book}$ is an example of a rejectable event.

2.3 The Role of Logic

Logic plays different roles in different formalisms surveyed in this paper. In [4], Temporal Logic serves as a specification medium only. It provides both the syntax and the semantics for the constraints. However, the workflow scheduler works directly with automata — the low-level representation of temporal constraints.

In contrast, logic is much more closely interwoven into the frameworks of [29] and [13]. In both formalisms, logic is a primary means of specifying the workflows. In addition, the workflow scheduler can be implemented as a particular strategy in the proof theory of the logic. In [29], the proof theory is based on the *residuation* operator and the scheduler uses this operator to make scheduling decisions. In [13], the scheduler depends on a preprocessing step after which the proof theory of Concurrent Transaction Logic [7] is employed directly to make scheduling decisions.

3 Modeling Workflows with Temporal Logic

In [4], Attie et al. proposed to model workflows as a set of intertask dependencies. Both local and global constraints (beginning of Section 2) can be modeled in this way and, therefore, the control flow graph is not represented explicitly.[2] The tasks in a workflow are described in terms of significant events. A typical event is the beginning or termination of a task, but it can also be sending an email to the boss, printing a report, etc.

When an event is received for execution, it is checked against every dependency and based on that the event might be accepted, rejected, or delayed and scheduled later. The dependencies are specified as formulae in Computational Tree Logic (CTL) [17]. The scheduler enforces these dependencies by converting them into automata and ensuring that the sequence of scheduled events is accepted by all these automata. In this way, the automata provide a low-level medium for the scheduler to work with, while the logic serves as a high-level specification medium.

This work does not explicitly deal with the verification issues, such as whether the given set of constraints implies some other constraints. Of course, the standard high-complexity model-checking techniques can be used here,

[2] It is unclear whether triggers can be naturally modeled using temporal logic.

but the interesting question is whether implication of workflow dependencies can be tested more efficiently due to the specialized form of these constraints.

3.1 Formalization

The formalization makes all the assumptions listed in Section 2: workflows are modeled as streams of significant events such as *start*, *precommit*, *commit*, and *abort*; the unique event assumption holds; and events can be delayable, rejectable, or forcible.

A workflow is specified as a set of dependencies over the events associated with the tasks. If $e_1, e_2, ..., e_n$ are the significant events associated with a number of tasks, then a dependency D involving these events is denoted as $D(e_1, e_2, ..., e_n)$. Computational Tree Logic (CTL) is used to specify these dependencies. For instance, the order dependency, $e_1 < e_2$, is specified in CTL as $\mathcal{A}\Box(e_2 \rightarrow \mathcal{A}\Box\neg e_1)$, *i.e.*, on every path it is always true that if e_2 occurs then e_1 will not occur later on any continuation of that path. A dependency, D, specified in CTL, is compiled into a finite state automaton A_D, which is a tuple $\langle s_0, S, \Sigma, \rho\rangle$, where:

- S is a set of states.
- s_0 is the initial state.
- Σ is a set of event expressions, which can have one of the following forms:
  - $a(e_1, ..., e_n)$, where $e_1, ..., e_n$ are events. This expression says that the events $e_1, ..., e_n$ are *accepted* by A_D and scheduled for execution. Each e_i is a significant event of some task.
  - $r(e_1, ..., e_n)$, where $e_1, ..., e_n$ are events. This expression says that the events $e_1, ..., e_n$ are *rejected* by A_D. The automaton A_D is constructed in such a way that the rejection takes place precisely when the execution of these events (in any order) would violate the dependency D.
  - $\sigma_1 \parallel ... \parallel \sigma_n$, where $\sigma_i \in \Sigma$. This expression specifies that the event expressions $\sigma_1, ..., \sigma_n$ are run *concurrently* in an interleaved fashion.
  - $\sigma_1; ...; \sigma_n$, where $\sigma_i \in \Sigma$. In this expression, the operations $\sigma_1, ..., \sigma_n$ are run in *sequence*.
- $\rho \subset S \times \Sigma \times S$ is the transition relation.

Figure 3(a) is an automaton for the occurrence dependency $e_1 \rightarrow e_2$. Here we use t_1 to denote the significant event of termination (*i.e.*, abort or commit) of task 1 and t_2 to denote the termination event for task 2. Symbols e_1 and e_2 are used to denote other, non-termination events. Because of the special semantics of termination events, no significant events from a task i can arrive once the event t_i has arrived and t_i must be scheduled last.

The symbol $|$ indicates choice — *either* event can cause the corresponding transition. This should be contrasted with the event combinator $\parallel$. For instance, an arc labeled with $a(e_1) \parallel a(e_2)$ means that *both* events, e_1 and e_2, must occur and the corresponding state transition can happen in one of the two ways: either by scheduling $a(e_1)$ first and $a(e_2)$ next or by scheduling these events in the reverse order.

The initial state in every automaton is denoted by i and the final state by f. Every *path* from the initial state to the final state corresponds to a way in which the dependency can be satisfied. Formally, for any dependency automaton, A_D, a *path* π is a sequence of event expressions $\sigma_1 \ldots \sigma_n$ such that there are states $s_1, s_2, \ldots, s_{n-1}, s_n$ in A_D, where

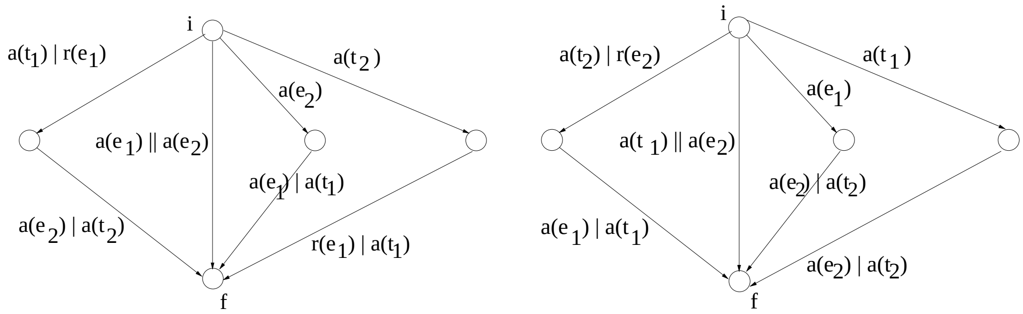
- s_1 is the initial state of A_D;
- s_n is a final state of A_D; and
- for each $i = 1, \ldots, n-1$: $(s_i, \sigma_i, s_{i+1}) \in \rho_D$, where ρ_D is the transition relation of A_D (*i.e.*, each σ_i is a legal transition from state s_i to s_{i+1} in A_D.

Figure 3(a) shows some event sequences that satisfy the dependency $e_1 \rightarrow e_2$:

- $r(e_1)\, a(e_2)$ — rejection of e_1 followed by acceptance of e_2.
- $a(t_1)\, a(e_2)$ — termination of task 1 followed by acceptance of e_2.
- $a(e_1)\, a(e_2)$ and $a(e_2)\, a(e_1)$ — because $a(e_1) \parallel a(e_2)$ is a label on one of the arcs, which means that executing e_1 and e_2 in any order can cause the corresponding transition.
- $a(e_2)\, a(t_1)$ — acceptance of e_2 followed by termination of task 1.
- $a(t_2)\, r(e_1)$ — termination of task 2 followed by rejection of e_1.
- $a(t_2)\, a(t_1)$ — termination of task 2 followed by termination of task 1.
- Other event sequences that satisfy the above dependency are $r(e_1)\, a(t_2)$, $a(t_1)\, a(e_2)$, and $a(e_2)\, a(e_1)$. Note that in the first case the event e_1 does not occur in the history, so the dependency $e_1 \rightarrow e_2$ is satisfied trivially. In the last case, the events occur in the reverse order. However, since both of them occur, the constraint is satisfied once again, as it only implies occurrence, not order.

Similarly, Figure 3(b) is an automaton for the order dependency $e_1 < e_2$. The sequence of events $a(t_1)\, a(e_2)$ is accepted by both automata, since each automaton has a path consistent with this sequence of events. However, the sequence $a(e_1)\, a(t_2)$ is accepted by the automaton for $e_1 < e_2$ only. This is because $a(e_1)\, a(t_2)$ does not correspond to a legal execution sequence in the automaton for the dependency $e_1 \rightarrow e_2$.

We can now define what it means for a sequence of events to be a legal execution. To simplify the matters, we augment the dependency automata with additional arcs. Namely, we add a self-loop to every state in each automaton. If n is a node in an automaton, then the corresponding self-looping arc has n as its beginning and end, and it is labeled by every event expression of the form $a(event)$ or $r(event)$ such that *event* is *not* mentioned on other outgoing arcs of n. (Note that if, say, $a(e_2)$ is mentioned on an outgoing arc of n then neither $a(e_2)$ nor $r(e_2)$ can occur on the self-looping arc.) The idea is that events that are not mentioned on these outgoing arcs leave the automaton in



(a) Automaton for the dependency $e_1 \rightarrow e_2$ (b) Automaton for the dependency $e_1 < e_2$

Fig. 3. Dependency automata

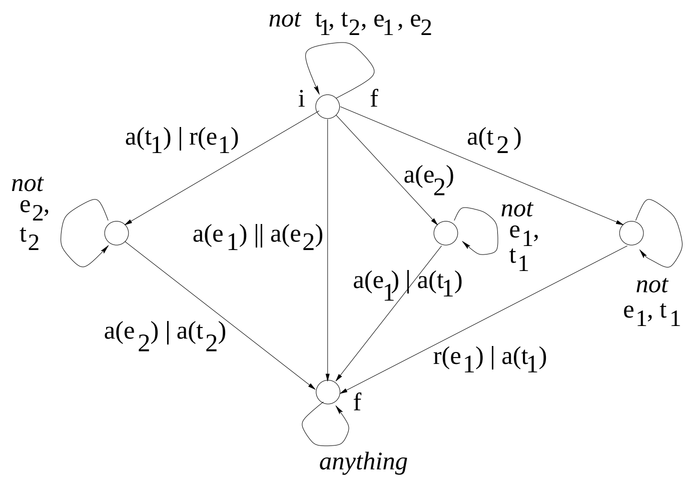


Fig. 4. Automaton of Figure 3(a) augmented with self-looping transitions

state n. In addition, we make the initial state of the automaton also into an accepting (final) state. The automaton of Figure 3(a) transformed in such a way is depicted in Figure 4. In this figure, a label such as "*not* e_2, t_2" means that the transition along that arc can be caused by any event expression that does not mention e_2 or t_2. For instance, neither $r(e_2)$ nor $a(e_2)$ can cause the transition, but $a(t_1)$ or $r(e_1)$ can.

We now define a sequence of events to be a *legal execution path* if it is accepted by *every* such augmented automaton. Note that due to the unique event assumption, an event can be mentioned in at most one event expression on the execution path.

3.2 Scheduling

In the previous subsection we have defined execution paths as sequences of event expressions. However, the scheduler receives sequences of events rather than event expressions. Thus, given a sequence of events, *seq*, the work of the scheduler is to find a legal execution path, π, such that the events mentioned in the expressions in π are all and only the events that occur in *seq*. If every automaton is of size N and there are m automata, then one can build a product automaton of size N^m. Unfortunately, this might be unacceptable for workflows that have many constraints.[3] To avoid this state explosion problem, the individual automata are checked at run-time, as explained below. The worst time complexity of run-time scheduling is still exponential. However, it is believed that the worst case does not occur in practice [4].

The *global state* of the scheduler is a tuple whose components are the local states of the dependency automata — one state per automaton. The *initial* global state is a tuple of the initial states of these automata. When an event, e, arrives, the algorithm tries to construct an event sequence, π, which is accepted by every *augmented* automaton, such that π includes e and (possibly) some of the events that have arrived previously but have not yet been scheduled (these are called *delayed* events).[4] In addition, each event on the path must occur at most once (*e.g.*, $a(e_2)$ and $r(e_2)$ count as multiple occurrences of the event e_2). If such a path cannot be found, then the scheduler delays the execution of the event e.

Consider the dependencies $e_1 \rightarrow e_2$ and $e_1 < e_2$ with the automata $A_{\rightarrow}$ and $A_{<}$, respectively, shown in Figure 3. Let $A'_{\rightarrow}$ and $A'_{<}$ be the augmentations of these automata. Augmentation for $A_{\rightarrow}$ is shown in Figure 4 and augmentation of $A_{<}$ is constructed similarly. Let e_1 be an event submitted to the scheduler. Since there is no path in *both* automata that begins by either accepting or rejecting e_1, the scheduling of e_1 has to be delayed. Now suppose the event e_2 is submitted to the scheduler. Two execution paths can be found in $A_{\rightarrow}$ that accept both e_1 and e_2: $a(e_2)a(e_1)$ and $a(e_1)a(e_2)$. The only path in $A_{<}$ that accepts both e_1 and e_2 is $a(e_1)a(e_2)$. However, in $a(e_1)a(e_2)$ the order of events is different from the path $a(e_2)a(e_1)$ in $A_{\rightarrow}$. Thus, the only legal execution path is $a(e_1)a(e_2)$ — the scheduler can execute e_1 followed by e_2 and satisfy both constraints.

---

[3] Observe that in this framework even the control flow graph is represented as a set of constraints, so the number of such constraints is, indeed, expected to be large.

[4] Note that e might not be mentioned in a *non*-augmented automaton, so there would be no guidance as to what to do when such an event arrives. An *augmented* automaton would simply discard such an event.

4 Modeling Workflows Using Event Algebra

In [29], Singh defines an algebra, which is suitable for reasoning about constraints over an incoming stream of events. This algebra is sufficiently expressive to represent very general temporal intertask dependencies, including control flow graphs. But conditions on transitions between tasks in such a graph cannot be expressed.[5] A scheduling algorithm starts with an expression that represents the entire set of constraints and then chips away at these expressions (or *residuates* in the terminology of [29]) as it schedules the arriving events.

While the event algebra is an elegant solution for the problem at hand, it is unclear whether it can model sub-workflows or be used to verify workflow properties such as whether a given set of constraints has redundancy in it or whether a constraint is implied by a set of constraints.

4.1 Formalization

Execution of a workflow relies on the notion of significant events produced by the tasks that comprise the workflow. Examples of such events are *start, precommit, commit*, and *abort.* A workflow is specified as a set of dependencies between these significant events. The dependencies are represented as event expressions in the algebra.

The set of symbols that represent significant events is denoted by Σ. This set does not need to be finite. An *atomic event expression* is either an event symbol from Σ or its *negation.* If $e \in \Sigma$, then its negation is represented as $\bar{e}$; it represents the assertion that e does not occur in the execution of the workflow. We will use lowercase letters to represent atomic events and capital letters for more complex event expressions.

The language of *event expressions*, denoted by $\mathcal{E}$, is defined as follows:

- $\Gamma = \{e, \bar{e} | e \in \Sigma\} \subseteq \mathcal{E}$.
  This just states that atomic events are event expressions. We use Γ to represent the set of atomic events.
- We distinguish two special event expressions : 0 and $\top$ in $\mathcal{E}$. The event 0 represents the event expression that is always false and the event $\top$ represents the expression that is always true.
- If $E_1, E_2 \in \mathcal{E}$, then $E_1 \cdot E_2 \in \mathcal{E}$. The operator "$\cdot$" denotes *sequencing, i.e.*, the event expression E_1 followed by the event expression E_2 (not necessarily immediately).
- If $E_1, E_2 \in \mathcal{E}$, then $E_1 + E_2 \in \mathcal{E}$. The operator "+" denotes *choice* or *disjunction.* The expression says that either the event expression E_1 must occur or E_2.

[5] Note that although [4] does not discuss scheduling in the presence of such conditions, they can at least be expressed in temporal logic.

- If $E_1, E_2 \in \mathcal{E}$, then $E_1|E_2 \in \mathcal{E}$. The operator "|" means *conjunction*. It denotes an event expression that represents both E_1 and E_2 occurring in any order.

The event algebra uses denotational style semantics where an event expression represents a set of legal traces. A *legal trace* (which we will often call just a *trace*) is a sequence of atomic events where

- Each event symbol occurs at most once in the same trace (the unique event assumption);
- An event and its negation cannot occur in the same trace; and
- For each $e \in \Sigma$, either e or $\bar{e}$ occurs in the trace.

Note that if $\bar{e}$ occurs in a trace then the exact placement of this symbol is immaterial: if an event does not occur in a trace then this remains true regardless where $\bar{e}$ was actually placed in the trace.

An event expression represents a constraint on the execution and the set of traces it represents are those that satisfy this constraint. For example, e is a constraint that says that the event e must occur, and the corresponding set of traces contains precisely those that have e in them. The event expression $e \cdot \bar{f}$ is a constraint that says that after e occurs then f cannot occur any more (*i.e.*, if f occurs at all, it must occur before e). The corresponding set of traces includes those that have e and either have no f or f occurs before e.

The set of traces (or *denotation*) for an event expression E is denoted by $[E]$. Given a set of atomic events, Γ, $U_\Gamma \subset \Gamma^* \cup \Gamma^\omega$ is the set of all finite (Γ^*) and infinite (Γ^ω) traces over the language Γ, *i.e.*, sequences of events that satisfy the three conditions given above.[6] The denotations of the various event expressions are defined as follows:

- $[e] = \{\tau \in U_\Gamma \mid e \in \tau,\ i.e.,\ e \text{ occurs in } \tau\}$
- $[0] = \emptyset$, that is no trace satisfies the expression 0.
- $[\top] = U_\Gamma$, that is every trace satisfies the expression $\top$.
- *Sequencing*: $[E_1 \cdot E_2] = \{\nu\tau \in U_\Gamma \mid \nu \in [E_1] \text{ and } \tau \in [E_2]\}$, that is the resulting trace is obtained by concatenation of the traces of E_1 and E_2.
- *Disjunction*: $[E_1 + E_2] = [E_1] \cup [E_2]$.
- *Conjunction*: $[E_1|E_2] = [E_1] \cap [E_2]$.

For an event expression $E \in \mathcal{E}$ and a trace $\tau \in U_\Gamma$, $\tau \models E$ denotes *satisfiability* of the event expression E by the trace τ, *i.e.*, the fact that $\tau \in [E]$.

Consider a travel workflow where one attempts to *buy* an airline ticket and *book* a car. The constraint is that either both tasks succeed or none succeeds. The other constraint is that *buy* cannot be canceled while *book* can. This workflow can be formulated in this algebra as the following set of dependencies:

[6] Note that if Σ is finite then there can be no infinite legal traces, due to the unique event assumption.

- $D_1 : \overline{start_{buy}} + start_{book}$: If *buy* starts then *book* must also start.
- $D_2 : \overline{commit_{book}} + \overline{commit_{buy}} + commit_{book} \cdot commit_{buy}$: If both *book* and *buy* commit then *book* commits before *buy*.
- $D_3 : \overline{commit_{buy}} + commit_{book}$: Task *buy* commits only if *book* commits.
- $D_4 : \overline{commit_{book}} + commit_{buy} + start_{cancel}$: If *book* commits and *buy* does not then start *cancel*.
- $D_5 : \overline{start_{cancel}} + \overline{commit_{buy}} | commit_{book}$: Start *cancel* only if *book* commits and *buy* does not.

This workflow is satisfied by several traces some of which are $start_{buy} start_{book} commit_{book} commit_{buy}$, $start_{book} start_{buy} start_{cancel}$, and $start_{book} start_{buy} commit_{book} commit_{buy}$.

4.2 Scheduling

Given a set of dependencies specified as a set of event expressions, the job of the scheduler is to find traces that satisfy the dependencies. The scheduler starts with an event expression that represents all dependencies. An incoming event is scheduled when this act is guaranteed to not break the dependencies regardless of which events will arrive in the future. The major insight here is that there is no need to record the past history of scheduled events. Instead, the information that is contained in the history and is relevant to the scheduler can be "recorded" in the *residual event expression* that remains to be satisfied by the future incoming event stream. We say "recorded" (in quotes) because — counter to the common intuition that recording of information leads to an increase of the data to be kept — recording of the relevant history leads to *simpler* residual event expressions.

This "recording" of history is done through the *residuation* operator. The state of the scheduler is represented by an event expression, D, which remains to be satisfied by the incoming stream of events. When a new event arrives, the residuation of D by e, denoted by D/e, is the new state of the scheduler.

Before giving a formal definition, we illustrate this notion by an example. Figure 5 shows the effect of residuation on the dependency $D_1 = \overline{start_{buy}} + start_{book}$ in the travel workflow discussed above. The dependency appears at the top of the figure and each node is labeled with an event expression (which might be a compound expression). Arcs are labeled by atomic event expressions. If the scheduler schedules an event that labels an arc, the result of the residuation would be the expression pointed to by the arc.

Suppose that the scheduler schedules the event $start_{buy}$. Since this implies that from now on all traces will contain this event, the traces represented by the $\overline{start_{buy}}$ will not be possible, so we can remove this part of D_1 and do not need to worry about it. Thus, D_1 is residuated to $start_{book}$. If however, the scheduler decides that *book* is not allowed to start (*i.e.*, it schedules $\overline{start_{book}}$ because of the need to satisfy some other constraint), then none of the traces that satisfy $start_{book}$ can occur, so we can remove that part of D_1. That

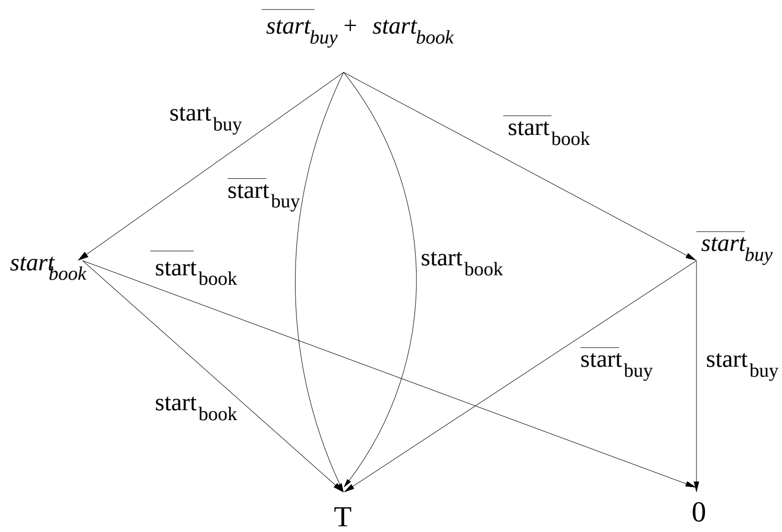


Fig. 5. Scheduler transitions for the dependency D_1 in the traveler workflow

is, $\overline{start}_{book}$ is scheduled then D_1 residuates to $\overline{start}_{buy}$, which becomes the dependency left to be satisfied. Informally, this means that if *book* is not allowed to start then the scheduler must ensure that *buy* is not allowed to start either. If the scheduler can schedule either $\overline{start}_{buy}$ or $start_{book}$ then the dependency D_1 is satisfied and it is residuated to $\top$. If a dependency cannot be satisfied then it residuates to 0. For example, suppose the current state of the scheduler is represented by the dependency $\overline{start}_{buy}$ and the event $start_{buy}$ arrives. Since there is no way to schedule this event (now or in the future) and still have the dependency satisfied, it is residuated to 0.

Formally, the residuation operator is defined as follows:

$v \in [E_1/E_2]$, where E_2 is an atomic event expression, if and only if for every trace $u \in [E_2]$ it holds that $uv \in [E_1]$.

If E has the form where neither "|" nor "+" occur under the scope of the sequencing operator "·", then the following rewrite rules provide an algorithm that computes residuation:

1. $0/e = 0$
2. $\top/e = \top$
3. $(E_1|E_2)/e = (E_1/e)|(E_2/e)$, where E_1 and E_2 are event expressions.
4. $(E_1 + E_2)/e = (E_1/e) + (E_2/e)$
5. $(e \cdot E)/e = E$, if $e, \bar{e}$ do not appear in the event expression E.
6. $(e' \cdot E)/e = 0$, if $e \neq e'$ and e occurs in E.
7. $(e' \cdot E)/e = 0$, if $\bar{e}$ occurs in E.
8. $E/e = E$, if e or $\bar{e}$ does not appear in the event expression E. This means that only the dependencies that mention the event e are relevant to residuation when e comes up for scheduling.

The residuation operator is *sound* and *complete* in the following sense. Let E be an event expression, e an event, and $E/e = E'$. Then there is a trace that satisfies E if and only if there is a trace that satisfies E'. Furthermore, τ' is a trace that satisfies E' if and only if $e\tau'$ (a sequence of events whose head is e and tail τ') is a trace that satisfies E.

A scheduler can now be constructed as follows. Let E be the initial event expression, which is a conjunction of all constraints. When an event, e, arrives, we compute $E' = E/e$. If $E' \neq 0$, the event is scheduled and E' becomes the new constraint that needs to be satisfied.

If $E' = 0$ due to the rule (7) then e cannot be scheduled and we have two choices. If e can be rejected, the scheduler does so and keeps E as its current state. If e is not rejectable, then the event stream cannot be scheduled and an error results.

If $E' = 0$ due to the rule (6) than e cannot be scheduled at this time, but it might be in the future. So, if e is delayable, it is delayed until such time when the dependency can be residuated by e to a non-0. Otherwise, if e is not delayable, the stream of events cannot be scheduled and an error results.

If E is in the form that permits the use of the above rewrite rules, the cost of a single scheduling operation is the cost of checking whether an event occurs in an expression and whether the result of residuation is 0. The former can be done in time logarithmic in the size of the expression. The complexity of the latter is linear in the size of E. If E is *not* in a proper form, we can achieve the desired form via a preprocessing step where "$\cdot$" is pushed into the event expression past the operators "+" and "|" using the following equivalences:

- $E_1 \cdot (E_2 + E_3) = E_1 \cdot E_2 + E_1 \cdot E_3$
- $E_1 \cdot (E_2 | E_3) = (E_1 \cdot E_2) | (E_1 \cdot E_3)$

This is analogous to the computation of a disjunctive/conjunctive normal form in classical logic, and is exponential in the size of E.

It can be shown that the above scheduling process terminates (albeit not always with success), because the size of the input event expression decreases monotonically.

5 Workflow Modeling Using Concurrent Transaction Logic

Concurrent Transaction Logic (CTR) [7] provides a uniform mechanism for modeling complex workflows, transforming them into more efficient workflows using logical equivalences, and for reasoning about workflow properties. The model theory of CTR provides a precise semantics both for workflows and global intertask dependencies, and serves as the yardstick of correctness for the transformation and verification algorithms. The proof theory of the logic can serve as a scheduler, which can execute workflow specifications.

5.1 Introduction to Concurrent Transaction Logic

The alphabet of CTR consists of:

- A set F of function symbols.
- A set P of predicate symbols.
- A set V of variables.

CTR terms are defined as in classical logic. A variable is a term. If f is a n-ary function symbol and $t_1, ..., t_n$ are terms, then $f(t_1, ..., t_n)$ is also a term.

CTR formulae are intended to represent transactions that execute by querying the underlying database state and modifying that state by adding or deleting facts. Informally, executing a transaction along a sequence of database states $\mathbf{D}_0$, ..., $\mathbf{D}_n$ (called a *path*) means that the transaction starts at state $\mathbf{D}_0$, changes it to state $\mathbf{D}_1$, then to $\mathbf{D}_2$, etc., terminating in state $\mathbf{D}_n$.

Formally, CTR formulae are defined as follows:

- Every atomic formula, $p(t_1, ..., t_n)$, where $p \in P$ and each t_i is a term, is a CTR formula.
  An atomic formula represents either an elementary update operation or a call to a complex transaction, whose behavior is defined via Horn-like rules.
- If ϕ is a CTR formula then so are the following formulae:
  - *Negation*: $\neg\phi$.
    A negated formula represents exactly those executions that are not executions of ϕ.
  - *Isolation*: $\odot\phi$.
    We shall see soon that execution of a CTR formula can interleave with the execution of other CTR formulas. That is, execution of ϕ can be interrupted to let another formula execute, and then resumed. The operator $\odot$ prevents this from happening, *i.e.*, $\odot\phi$ represents "uninterrupted" executions of ϕ (or "isolated" executions, if we use the terminology of database transaction processing).
  - *Quantification*: $(\forall X)\phi$.
    Executing such a formula along a path, π, means that π is an execution path for every formula that is obtained from ϕ by instantiating X with a ground (*i.e.*, variable-free) term.
- If ϕ and ψ are CTR formulae then so are:
  - *Classical Conjunction*: $\phi \wedge \psi$.
    This formula says, execute ϕ in such a way that the execution path will also be a valid execution of ψ (or, equivalently, execute ψ so that it will also be a valid execution of ϕ). We shall see later that classical conjunction forms the basis for representing constraints on the executions of workflows. Typically ϕ would represent a workflow and ψ a constraint.

- *Serial Conjunction*: $\phi \otimes \psi$.
  Intuitively, this formula says, execute ϕ and then ψ. Serial conjunction forms the basis for representing sequential composition of tasks in a workflow.
- *Concurrent Conjunction*: $\phi | \psi$.
  Concurrent conjunction is used to specify concurrent, interleaved execution of sub-workflows. A valid execution of the above formula could be a path where one subworkflow, say, ϕ starts. This execution may be interrupted by an execution of ψ. Execution of ψ can also be interrupted and ϕ may be resumed. The resumed execution of ϕ may again be interrupted and ψ resumed, etc.

We omit other operators, such as $\diamond$ and $\Box$, which are not used for workflow modeling. Additional, convenience operators can be defined similarly to classical logic:

- $\phi \vee \psi \equiv \neg(\neg\phi \wedge \neg\psi)$.
- $\phi \leftarrow \psi \equiv \phi \vee \neg\psi$.
- $\exists\phi \equiv \neg\forall\neg\phi$.

The following CTR formula illustrates the use of some of the above connectives:

$$b \otimes ((d \otimes cond_3 \otimes h) \vee e) \otimes j$$

This formula happens to represent the part of the workflow graph in Figure 2, page 5, which begins with activity b and ends with activity j. We will talk more about modeling of workflows in CTR in Section 5.

The semantics of database states and state transitions in CTR is defined using a pair of oracles. Intuitively, a state is a set of data items.

- A *data oracle*, O^d, is a mapping from states to sets of first order formulae. If D is a state, $O^d(D)$ represents the set of formulae that are true in that state.
- A *transition oracle*, O^t, is a mapping from pairs of states to sets of atomic formulae. If $b \in O^t(D_1, D_2)$, then b is interpreted as an update that changes state D_1 into D_2.

For example, if D_1 is a database state where the formulas p and q are true then $p, q \in O^d(D_1)$. Also, let $insert(r)$ and $delete(p)$ be atomic formulas which insert and delete propositions r and p respectively. Let D_2 be the database state where the formulas $p, q,$ and r are true and D_3 be the database state where q and r are true. Then $insert(r) \in O^t(D_1, D_2)$ and $delete(p) \in O^t(D_2, D_3)$. Note that in this example we have defined *a concrete* data oracle and a transition oracle. The propositions *insert* and *delete* are given special meaning by that particular transition oracle; they are *not*, however, special keywords of CTR. Some other oracle might use a different set of propositions for elementary updates, and the semantics of those propositions can be completely different [8] as well.

The semantics of CTR formulae are defined over *multi-paths*. A multi-path is a finite sequence of *paths*. A path is a finite sequence of database states which represents the execution of a formula ϕ. A path must have at least one state and a multi-path at least one path. In a multi-path, every constituent path represents a period of continuous execution of a transaction. For instance, if $D_1, D_2, ..., D_8$ are database states, then $\pi = \langle D_1 D_2 D_3, D_4 D_5, D_6 D_7 D_8 \rangle$ is a multi-path comprised of three paths, which represents the execution of a formula. The paths that constitute π are separated with commas. Thus, the first path, $D_1 D_2 D_3$ in π represents the first burst of continuous execution of ϕ in which the transaction changes the initial state, D_1 to D_2 and then to D_3. This initial burst is interrupted by the execution of other transactions, which (possibly through a long sequence of changes) leaves the database in state D_4. At this point ϕ resumes, changes the database state to D_5, and is interrupted again. While ϕ is suspended, other transactions change the database state to D_6. At this time ϕ wakes up again, and its execution changes the state to D_7, D_8, and terminates.

The semantics of CTR formulae is given by *multi-path structures*, which determine the truth-value of each formula on the different multi-paths. The intuitive meaning of a formula that is true on a multi-path is that this formula can execute along this multi-path, changing the underlying database state as specified in that multi-path.

Formally, a *multi-path structure*, M, is a mapping from multi-paths to the classical first-order semantic structures that are used to interpret formulas in predicate calculus. Thus, given a multi-path, π, $M(\pi)$ is a first-order semantic structure. This mapping is required to be consistent with the data and transition oracle in a natural way:

- *Data oracle consistency*: For 1-paths of the form $\langle D \rangle$, $M(\langle D \rangle) \models O^d(D)$ Intuitively this means that formulas in $O^d(D)$ are defined to have valid executions over the path $\langle D \rangle$. (Note that $\models$ is well-defined here because $M(\langle D \rangle)$ is, by definition, a first-order semantic structure.)
- *Transition oracle consistency*: For 2-paths of the form $\langle D_1, D_2 \rangle$, $M(\langle D_1\, D_2 \rangle) \models O^t(D_1, D_2)$.
  Intuitively this means that the elementary transitions in $O^t(D_1, D_2)$ are defined to have valid executions over the path $\langle D_1\, D_2 \rangle$. (Note that $\models$ is, again, well-defined, since $M(\langle D_1\, D_2 \rangle)$ is a regular first-order structure.)

If M is a multi-path structure and π a multi-path, then the satisfaction of a formula ϕ on π in structure M is denoted by $M, \pi \models \phi$ and is defined as follows:

- *Atomic Formula*: $M, \pi \models p(t_1, ..., t_n)$, if and only if $M(\pi) \models p(t_1, ..., t_n)$ for any atomic formula $p(t_1, ..., t_n)$. Intuitively, the truth of an atom

$p(t_1, ..., t_n)$ on a multi-path π means that a transaction p can execute along π when invoked with the arguments $t_1, ..., t_n$.

- *Negation*: $M, \pi \models \neg\phi$, if and only if it is not the case that $M, \pi \models \phi$.
- *Isolation*: $M, \pi \models \odot\phi$, if and only if $M, \pi \models \phi$ and π is a path (not a multi-path). Intuitively, this means execute ϕ in isolation without inter-leaving with the execution of other formulae.
- *Quantification*: $M, \pi \models \forall X.\phi$, if and only if $M, \pi \models \phi[X/a]$ for every assignment of a ground term to the variable X. Here $\phi[X/t]$ denotes ϕ with all free occurrences of the variable X replaced by the term t.
- *Classical Conjunction*: $M, \pi \models \phi \wedge \psi$, if and only if $M, \pi \models \phi$ and $M, \pi \models \psi$.
- *Serial Conjunction*: $M, \pi \models \phi \otimes \psi$, if and only if $M, \pi_1 \models \phi$ and $M, \pi_2 \models \psi$, for some multi-paths π_1, π_2, and $\pi = \pi_1 \bullet \pi_2$.
  Here $\pi = \pi_1 \bullet \pi_2$ denotes the concatenation of the multi-paths π_1 and π_2. For instance, $\langle D_1D_2D_3, D_4D_5, D_6D_7D_8\rangle \bullet \langle D_9D_{10}, D_{11}D_{12}\rangle$ is $\langle D_1D_2D_3, D_4D_5, D_6D_7D_8, D_9D_{10}, D_{11}D_{12}\rangle$.
- *Concurrent Conjunction*: $M, \pi \models \phi|\psi$, if and only if $M, \pi_1 \models \phi$ and $M, \pi_2 \models \psi$, for some multi-paths π_1, π_2, and with an inter-leaving in $\pi_1||\pi_2$ that reduces to π, as explained below.
  Here $\pi_1||\pi_2$ denotes an *interleaving* of the multi-path π_1 with the multi-path π_2, which is a multi-path that consists of paths drawn from π_1 and π_2 and the order of those paths in the interleaving is consistent with their order in π_1 and π_2. For instance, one interleaving of the multi-path $\langle D_1D_2D_3, D_4D_5, D_6D_7D_8\rangle$ with $\langle D_9D_{10}, D_{11}D_{12}\rangle$ is $\langle D_1D_2D_3, D_9D_{10}, D_4D_5, D_{11}D_{12}, D_6D_7D_8\rangle$. Another is $\langle D_1D_2D_3, D_9D_{10}, D_4D_5, D_6D_7D_8, D_{11}D_{12}\rangle$.
  A multi-path π *reduces* to another multi-path if some adjacent paths in π can be spliced into one path because the end of the preceding path coincides with the start of the next path. For instance, none of the paths above reduces to any other path (except itself). But the following multi-path $\langle D_1D_2D_3, D_3D_4, D_5D_6D_7, D_7D_8, D_9D_{10}\rangle$ reduces to the path $\langle D_1D_2D_3D_4, D_5D_6D_7D_8, D_9D_{10}\rangle$.

A multi-path structure M is a *model* of a formula ϕ, denoted by $M \models \phi$, if and only if $M, \pi \models \phi$ for every multi-path π.

The following example illustrates how updates can be combined with queries to define complex transactions using CTR. It also illustrates the role of oracles for defining state queries and elementary transitions as well as the model theory.

Example 1 (Relational Database Transactions). For this example we will use *relational* data and transition oracles, which encapsulate queries and updates performed on relational databases. They are defined as follows:

Relational oracles: A *relational database state* is a set of ground atomic formulas, D. For each relation name p in the database, we define the *relational* *data oracle* as $p(\overline{x}) \in O^d(D)$ iff $p(\overline{x}) \in D$. The *relational transition oracle* defines, for each variable-free atomic formula $p(\overline{x})$, a pair of new propositions, $insert(p(\overline{x}))$ and $delete(p(\overline{x}))$, representing the insertion of the atom $p(\overline{x})$ and its deletion, respectively. Formally, $insert(p(\overline{x})) \in O^t(D_1, D_2)$ iff $D_2 = D_1 \cup \{p(\overline{x})\}$ and $delete(p(\overline{x})) \in O^t(D_1, D_2)$ iff $D_2 = D_1 - \{p(\overline{x})\}$.

Note that due to the consistency requirement for transition oracles, if M is an m-path structure then the first-order semantic structure $M(\langle D_1, D_2 \rangle)$ must interpret the predicates `insert` and `delete` that are defined by the oracle. However, it can interpret additional predicates as well. For instance, if we have a rule

$$foobar(X) \leftarrow delete(X).$$

and M is a model of this rule then $M(\langle D_1, D_2 \rangle)$ must interpret the predicate `foobar` as well.

Consider the following formula

$$\phi \;=\; insert(a) \;\otimes\; (insert(b) \mid (b \otimes delete(a)))$$

The possible models for ϕ can be computed from the models of the components of ϕ as follows:

1. By the definition of the relational transition oracle: $\langle \{\}\ \{a\} \rangle \models insert(a)$, $\langle \{a\}\ \{a,b\} \rangle \models insert(b)$, $\langle \{a,b\}\ \{b\} \rangle \models delete(a)$; $\langle \langle \{a,b\} \rangle \rangle \models b$ by the definition of the relational data oracle;
2. $\langle \{a,b\}\ \{b\} \rangle \models (b \otimes delete(a))$; by the definition of $\otimes$;
3. $\langle \{a\}\ \{a,b\}\ \{b\} \rangle \models (insert(b) \mid (b \otimes delete(a))$; by the definition of $\mid$;
4. $\langle \{\}\ \{a\}\ \{a,b\}\ \{b\} \rangle \models insert(a) \;\otimes\; (insert(b) \mid (b \otimes delete(a))$; by the definition of $\otimes$.

Executional entailment ties the semantics described above to the notion of execution. If P is a set of formulae and D_0 is a database state then $P, D_0 \text{---} \models \phi$ is true if and only if there are states $D_1, ..., D_n$ such that $M, \langle D_0 D_1 ... D_n \rangle \models \phi$ for every multi-path structure M that is a model of P. Note that here we are interested in an uninterrupted execution, because the multi-path has only one path in it. Informally, this means that the formula ϕ can execute successfully starting from the database state D_0 and may change the database state in the process.

As we shall see, ϕ can be thought of as a workflow with tasks being composed sequentially and in parallel. P could be empty or it can be a set of rules that define the behavior of the individual tasks in the workflow. If $P, D_0 \text{---} \models \phi$ holds, it means that the workflow can execute along some path starting at state D_0. One of the most interesting properties of CTR is that its proof theory constructs this execution path and in this sense it can be said to *execute* ϕ.

We shall not go into the details of the proof theory of CTR (see [7]), but will illustrate it via an example. A sound and complete proof theory exists for the *concurrent-Horn* subset of the logic. A *concurrent-Horn goal* (which is used either as a query or a rule body) is:

- Any atomic formula.
- If ϕ and ψ are concurrent-Horn goals, then so are:
  - $\phi \otimes \psi$.
  - $\phi | \psi$.
  - $\phi \vee \psi$.
- If ϕ is a concurrent-Horn goal, then so is $\odot\phi$.

Concurrent Horn goals are used here as queries or rule bodies. This is slightly different from some conventions where goals are of the form $\leftarrow$ *body*. The difference is, however, in exposition, not substance.

A *concurrent-Horn rule* is a CTR formula of the form *head* $\leftarrow$ *body*, where *head* is an atomic formula and *body* is a concurrent-Horn goal. The concurrent-Horn subset of CTR consists of concurrent-Horn rules and concurrent-Horn goals. Given a concurrent-Horn goal, the proof theory identifies the set of subformulae that can execute at any given time and applies inference rules to simplify the goal until either the deduction succeeds or fails.

To illustrate, assume that our database states are simply sets of propositional constants and the oracles are relational, as defined in Example 1. Let program P contain the following rules:

$$p \leftarrow insert(a) \otimes q \otimes delete(p)$$
$$r \leftarrow insert(q) \otimes a \otimes insert(s)$$

and consider the goal $(p \otimes s) \mid r$. In this example, the goal can be viewed as a workflow and p, r, and s as its subworkflows.

Suppose we want to find out if it can be executed beginning with the database state $\{p\}$, *i.e.*, whether the executional entailment

$$P, \{p\} \text{---} \models (p \otimes s) \mid r$$

is true. The proof theory proceeds by trying to execute either side of $|$. Let us choose r, which we can expand using the second rule and obtain the following goal:

$$(p \otimes s) \mid (insert(q) \otimes a \otimes insert(s))$$

Let us proceed with the execution of the right side of the formula and execute its first literal, $insert(q)$. This will reduce the goal to

$$(p \otimes s) \mid (a \otimes insert(s))$$

and change the database state to $\{p, q\}$. We can try to continue executing the right side of the formula, which requires checking if the proposition a is true in the current state. It is not. In Prolog, the entire goal would fail (*i.e.*, found to be false), but in CTR we have to wait and see if a might become true as a result of other, concurrent activities. Not being able to proceed with the right side of the goal, we switch our attention to the left side and expand p using the first rule:

$$(insert(a) \otimes q \otimes delete(p) \otimes s) \mid (a \otimes insert(s))$$

Continuing with the left side, we can execute $insert(a)$, which causes a state change to $\{p, q, a\}$. The goal now reduces to

$$(q \otimes delete(p) \otimes s) \mid (a \otimes insert(s))$$

Note that now a has become true and we can proceed with the right side of the formula. Having checked that a is true, we can delete it from the goal. We can also check q, which happens to be true, and delete it from the goal. Neither operation causes a state change. The resulting goal is

$$(delete(p) \otimes s) \mid insert(s)$$

We can now execute $delete(p)$, causing a state transition to $\{q, a\}$. We cannot proceed with the left side of the formula, because s is not true in the current state, but we can proceed with the right side, inserting s. Thus, the new state becomes $\{q, a, s\}$ and the goal reduces to the query s. Since it is true in the current state, the executional entailment has been established. By tracing the sequence of state changes that occurred during the proof, we can reconstruct the execution path of the goal: $\{p\}$, $\{p, q\}$, $\{p, q, a\}$, $\{q, a\}$, $\{q, a, s\}$.

5.2 Modeling Workflows as $\mathcal{CTR}$ Goals

CTR can model workflows at several levels. CTR goals are expressive enough to model complex control flow graphs and rules can be used to model sub-workflows. The head of a rule can be seen as a compound task and the body of a rule (which is a CTR goal) is a control flow graph that represents the workflow that defines that compound task.

The overall idea behind using CTR for modeling workflow control graphs is very simple. Propositional constants can be used to represent individual tasks, the connective $\otimes$ represents sequential composition of tasks and $\mid$ can be use to combine tasks in parallel. In addition, classical disjunction, $\vee$, represents non-deterministic choice and transition conditions between tasks can be modeled as queries. For instance, consider the control flow graph in Figure 2 on page 5, which includes both sequential and concurrent composition of tasks as well as transition conditions. It can be represented as the concurrent-Horn goal as follows:

$$\begin{aligned} a \otimes \big((cond_1 \otimes b \otimes ((d \otimes cond_3 \otimes h) \vee e) \otimes j) \mid \\ (cond_2 \otimes c \otimes ((f \otimes i \otimes cond_4) \vee (g \otimes cond_5))) \big) \otimes k \end{aligned} \quad (1)$$

This goal represents a workflow control graph, which is part of the workflow specification. The remaining part, intertask dependencies, is also specified using CTR, as explained later. We shall see that CTR can be used not only to specify workflows, but also reason and schedule them.

One can also represent data flow using predicates with variables in the CTR goals, but we will not get into these aspects.

5.3 Using *CTR* to Schedule and Verify Workflows

A uniform framework for specifying, verifying and scheduling workflows was proposed in [13]. A workflow is modeled as a control flow graph, which is specified as a concurrent Horn CTR goal G, and a set of global dependencies, D. Note that unlike the approaches based on Temporal Logic and algebra, here we distinguish between local precedence constraints, which are represented in the control flow graph and global constraints (such as those listed in the third column in Figure 2 on page 5) which cannot be easily represented in this way.

The entire workflow is represented as a conjunction $G \wedge D$. Recall that in CTR such a conjunction means, execute the workflow G so that all the constraints in D will be satisfied. The question here is how to execute such a workflow. Indeed, we saw that the proof theory of CTR can execute concurrent-Horn goals, but the above specification is not such a goal due to the classical conjunction $\wedge$. We could try to execute the G-part of the workflow constantly checking that the D part is satisfied, but this would cause much backtracking at run time, which is undesirable.

It turns out, however, that under certain assumptions we can find an equivalent CTR formula, G', which happens to be concurrent-Horn and thus can be executed by the proof theory (and without backtracking). *Equivalence* here means that G and G' have the same models, as in most other logics. We can view the process of finding G' as scheduling, because G' can be viewed as a concise representation of all possible valid schedules. Thus, there is an important difference between the nature of scheduling in CTR and the approaches described in Sections 3 and 4. In the latter approaches, the scheduler is *passive* — it is waiting for the events to arrive during workflow execution. In CTR on the other hand, the scheduler is *pro-active*: it compiles an original workflow specification, $G \wedge D$, into one (G') where scheduling decisions become trivial.

The size of G' is *linear* in the size of G but *exponential* in the size of the dependencies D. Since the size of the dependency set is usually much smaller than the size of the control flow graph, verification of the properties of G using this method is more efficient than standard model-checking techniques which are worst-case exponential in the size of the control flow graph.

Thus, the approach of [13] causes certain blowup in the size of the control flow graph (which might be expensive, but not prohibitively so, because it is exponential only in the size of the global dependencies), but run-time scheduling takes linear time in the depth of that graph. The temporal logic based approach, [4], faces a similar choice: pay exponential price and compile time to enable linear-time scheduling or do nothing at compile time and incur exponential complexity during scheduling. Unfortunately, the first choice is prohibitively expensive for large workflows (exponential in the size of *both* the control graph and global dependencies). Although the second choice (paying at run-time) has exponential worst-case complexity, it is believed that the average complexity is "not so bad." The algebraic approach, [29], requires a preprocessing step (conversion to DNF) that can increase the size of the constraints exponentially. Since these constraints represent both the local control flow and the global constraints, the worst case complexity is rather high. The algebraic approach also incurs certain run-time overhead, as discussed in Section 4.2.

Formalization. As in [29,4], workflows are modeled in terms of significant events of the workflow tasks such as *start, precommit, commit*, and *abort*. As with other approaches considered in this paper, the *unique event property* is assumed to hold.

The events are specified as propositions drawn from a set of events, denoted as $Event$. A special proposition *path*, defined as $\phi \vee \neg\phi$ for any CTR formula ϕ, is used to denote the counterpart of *true* in classical logic. That is, *path* is true on all execution paths. For convenience, we use $\bigtriangledown\phi$ as a shorthand for $path \otimes \phi \otimes path$. If G is a CTR goal that represents a workflow graph, then $G \wedge \bigtriangledown\phi$ means that G must be executed in such a way that ϕ is true somewhere on the execution path. Thus, if ϕ is an event, $\bigtriangledown\phi$ is a constraint that the event must occur some time during the execution. The dependencies that can be specified in this framework are:

- *Primitive Dependencies*: $\bigtriangledown e$ and $\neg\bigtriangledown e$, where $e \in Event$.
  The first constraint says that e must occur, while the second says that it should not occur.
- *Serial Dependencies*: if $d_1, d_2, ..., d_n$ are primitive dependencies of the form $\bigtriangledown e_i$, then $d_1 \otimes \ldots \otimes d_n$ is a serial dependency.
  For instance, $\bigtriangledown e \otimes \neg\bigtriangledown f \otimes \bigtriangledown g$ is a constraint that says that the execution consists of three parts. In the first, the event e must occur. This should follow by an execution where f does not occur. In the third phase, g must occur.
- *Complex Dependencies*: if D_1, D_2 are dependencies, then so are $D_1 \vee D_2$ and $D_1 \wedge D_2$.

The logic is expressive enough to model both *order* ($e_1 < e_2$), and *occurrence* ($e_1 \rightarrow e_2$) dependencies. The order dependency is modeled as $\neg\bigtriangledown e_1 \vee \neg\bigtriangledown e_2 \vee (\bigtriangledown e_1 \otimes \bigtriangledown e_2)$. The occurrence dependency is modeled as $\neg\bigtriangledown e_1 \vee \bigtriangledown e_2$. It can be proved that the set of dependencies is closed under negation.

Scheduling and verification. Given a workflow, G, specified as a concurrent-Horn goal, and a set of dependencies, D, it can be verified whether G is consistent with D. If so, the workflow can be scheduled in linear time in the depth of G (after some transformation, which is described below). In addition, it can be checked whether the workflow specification entails some other constraint, ϕ. To this end, one simply needs to check whether $G \wedge D \wedge \neg\phi$ is consistent.

The main technique rests on a transformation, which compiles the dependencies, D, into the control flow graph G. The result is another control flow graph, G', which is logically equivalent to $G \wedge D$. Thus, the transformation is sound and complete. Although, as mentioned above, G' is worst-case exponential in the size of D, this is not a serious problem in practice. First, D is much smaller than G. Second, for some constraints the blowup is only polynomial.

The constraints are compiled into G using the procedure *Apply*. If G is a concurrent-Horn goal and d is a dependency, then $Apply(d, G)$ is defined to yield a CTR goal, G', which is equivalent to $G \wedge d$. This is done in the following way:

- *Compiling primitive dependencies*: If $e_1, e_2 \in Event$ and G_1, G_2 are concurrent-Horn goals then:
  - $Apply(\nabla e_1, e_1) \equiv e_1$
  - $Apply(\nabla e_1, e_2) \equiv \neg path$, if $e_1 \neq e_2$
  - $Apply(\neg \nabla e_1, e_1) \equiv \neg path$
  - $Apply(\neg \nabla e_1, e_2) \equiv e_2$, if $e_1 \neq e_2$
  - $Apply(\nabla e_1, G_1 \otimes G_2) \equiv Apply(\nabla e_1, G_1) \otimes G_2 \vee G_1 \otimes Apply(\nabla e_1, G_2)$
  - $Apply(\neg \nabla e_1, G_1 \otimes G_2) \equiv Apply(\neg\nabla e_1, G_1) \otimes Apply(\neg \nabla e_1, G_2)$
  - $Apply(\nabla e_1, G_1|G_2) \equiv Apply(\nabla e_1, G_1)|G_2 \vee G_1|Apply(\nabla e_1, G_2)$
  - $Apply(\neg \nabla e_1, G_1|G_2) \equiv Apply(\neg\nabla e_1, G_1)|Apply(\neg \nabla e_1, G_2)$
  - $Apply(\sigma, \odot G_1) \equiv \odot Apply(\sigma, G_1)$, where σ is ∇e_1 or $\neg \nabla e_1$
  - $Apply(\sigma, G_1 \vee G_2) \equiv Apply(\sigma, G_1) \vee Apply(\sigma, G_2)$
- *Compiling serial dependencies*: If $e_1, e_2 \in Event$ and G is a concurrent-Horn goal then,
  - $Apply(\nabla e_1 \otimes \nabla e_2, G) \equiv Apply(\nabla(e_1) \otimes send(\varepsilon) \wedge Apply(receive(\varepsilon) \otimes \nabla(e_2), G))$, where ε is a new constant. The *send* and *receive* primitives can be defined in CTR as part of a transition oracle (just like *delete* and *insert*), so that their semantics would be such that $receive(\varepsilon)$ is true if and only if $send(\varepsilon)$ has been previously executed [7].
- *Compiling complex dependencies*: If D_1, D_2 are complex dependencies and G is a concurrent-Horn goal:
  - $Apply(D_1 \vee D_2, G) \equiv Apply(D_1, G) \vee Apply(D_2, G)$
  - $Apply(D_1 \wedge D_2, G) \equiv Apply(D_1, Apply(D_2, G))$

Compiling the dependencies D into the original goal G yields either a new concurrent-Horn goal G' or $\neg path$. The workflow control flow graph is inconsistent with the set of dependencies if the result of the *Apply* procedure is $\neg path$. Even if *Apply* does not yield $\neg path$, the result might still be inconsistent or contain redundancy because G' can have subformulae where the *send/receive* primitives form a circular wait. These regions in G' are known as *knots*. A procedure *Excise* (which we will not describe here) takes the result of the *Apply* procedure and returns either a knot-free concurrent-Horn goal or $\neg path$.

The procedures *Apply* and *Excise* can be used to check workflows for consistency and verify some other properties as follows. Given a workflow specification, G, and a set of dependencies, D, the workflow specification is inconsistent if and only if $Excise(Apply(D, G)) \equiv \neg path$. A property Φ (represented as a constraint) is satisfied by every execution of the workflow if and only if $Excise(Apply(\neg\Phi \wedge D, G)) \equiv \neg path$.

A consistent workflow can be scheduled by simply using the proof theory of CTR on the goal obtained by computing $Excise(Apply(D, G))$. This will find a suitable execution path for the workflow G while obeying the constraints in D.

In general, workflow scheduling in CTR is NP-complete if both compile-time and run-time phases are taken into the account [13]. A similar NP-completeness result was obtained in [32]. However, after the compilation is done, unlike [4,29], the scheduler need not make any run-time decisions since all the dependencies are pre-compiled into the control-flow graph. As a result, each scheduling step takes constant time and the entire schedule can be constructed in linear time in the size of the *original* control flow graph.

Extensions. As we have seen, unlike the other framework discussed in this paper, CTR can model workflows whose control flow graphs have transition conditions on the arcs of the control flow graph. (See, for example, how the transition conditions of the graph in Figure 2 are represented in the CTR formula (1).) The compilation technique of [13] described earlier is still applicable to such workflows. In particular, it eliminates the need for run-time scheduling decisions due to the global constraints and it can also detect inconsistency between these constraints and the control flow graph. Scheduling can still be performed by the proof theory of CTR. However, due to the presence of transition conditions, scheduling is no longer linear as it might require backtracking over some previously scheduled tasks.

CTR-based modeling can also be extended in the direction of workflows that must execute under various aggregate or resource allocation constraints. Examples of such constraints are cumulative time and cost constraints on the execution of a workflow such as travel reservation, or constraints on the allocation of machines in a job-shop scheduling workflow. One way to model such problems is to extend CTR to *Constraint CTR*, which adds constraint-solving capability a la Constraint Logic Programming. This direction is pursued in [28].

Another extension of CTR, which can potentially be useful to model Web services, is to add certain game-theoretic capabilities to the logic. As a result, it becomes possible to model workflows that include *non*-cooperating (or even adversarial) activities. For instance, the Web services standards such as UDDI, WSDL, and BPEL4WS [27,9,11] make it possible for anybody to publish an *aggregate* Web service, *i.e.*, a workflow composed out of Web services published by others. The constituent services might uphold the various contracts and laws, but they cannot be assumed to act in the best interests of the aggregator. Therefore, there is a need to be able to specify contracts and verify that the goals of the aggregate workflow will be met provided that the constituent workflows follow the letter of the contracts. A step in this direction was made in [14].

Expressiveness. In [6], Bonner investigated the expressive power of concurrent-Horn CTR. The complexity of the logic depends on what kind of atomic formulas are allowed. Four kinds of atomic formulae have been identified:

- *Querying*, $p(x)$: Check if $p(x)$ is in the database.
- *Emptiness Checking*, $empty(p)$: Check if the database contains no atom of the form $p(x)$.
- *Insertion*, $insert(p(x))$: Insert atom $p(x)$ into the database.
- *Deletion*, $delete(p(x))$: Delete atom $p(x)$ from the database.

The semantics of these four elementary operations can be defined in terms of the executional entailment.

- *Querying*: $P, D_1 D_2 \models p(x)$ if and only if $D_1 = D_2$ and $p(x) \in D_1$.
- *Emptiness Checking*: $P, D_1 D_2 \models empty(p)$ if and only if $D_1 = D_2$ and $p(x) \notin D_1$ for all x.
- *Insertion*: $P, D_1 D_2 \models insert(p(x))$ if and only if $D_2 = D_1 \cup \{p(x)\}$.
- *Deletion*: $P, D_1 D_2 \models delete(p(x))$ if and only if $D_2 = D_1 - \{p(x)\}$.

The expressiveness of the logic depends on the complexity of the *transactions* accepted by it. A database *schema* S is a finite set of predicate symbols. The *domain* of a database D, denoted by $dom(D)$, is the set of constant symbols in it. A database transaction $\langle S_1, S_2 \rangle$ is a binary relation on database states D_1, D_2 with respective schemas S_1, S_2. Informally, a transaction changes a database D_1 with schema S_1 to a database D_2 with schema S_2. A transaction T is *safe* if $dom(D_1) \supseteq dom(D_2)$ for every pair $\langle D_1, D_2 \rangle$ in T. Given a set of formulae P and a goal ϕ, the logic *expresses* a transaction T if $\langle D_1, D_2 \rangle \in T$ if and only if $P, D_1 D_2 \models \phi$. The *data complexity* of the logic is the complexity of the most complex transaction accepted by the logic. The logic is *data complete* for a complexity class if it can accept all transactions in that class. For instance, if a logic is data complete for NP then it can express all NP-complete transactions. It turns out that depending on what atoms are allowed the expressive power of the logic can vary greatly.

- If *queries* are the only atomic formulae, the logic is data complete for $PTIME$.

- If *queries* are allowed and either *insertion* or *deletion*, the data complexity is $PSPACE$.
- If *queries* and both *insertion* and *deletion* are allowed, the logic is data complete for RE.
- Without *emptiness checking*, the logic can only accept *monotonic* goals. A monotonic goal has the property that if it terminates and commits starting from a database state D, then it also terminates and commits when started from any database state containing D. With the addition of *emptiness checking*, the logic becomes non-monotonic. It can express every safe transaction in RE.

The effect of concurrency and recursion on the data complexity of the logic is as follows:

- The logic is data complete for $EXPTIME$ without the $|$ operator (i.e. a sequential logic).
- Without recursion — but with concurrency — the data complexity is *LOGSPACE*.
- If only *sequential tail recursion* is allowed, the logic is data complete for $PSPACE$. A rule exhibits sequential tail recursion if it has the form $p \leftarrow \psi \otimes q$, where ψ is a goal and q is the only atom in the body which is mutually recursive with p.

[6] further develops a subset of concurrent-Horn CTR, called *fully-bounded*. It retains a wide range of workflow modeling capabilities and yet has low complexity. The idea behind full-boundedness is:

- Every recursive call to a predicate must remove a tuple from a base relation.
- Tuples that are removed from a relation must not find their way back into the relation.

To find out if a set of rules is fully-bounded, a data flow graph is constructed. This graph keeps track of the flow of tuples between different relations at each level of recursion. If the graph is acyclic, then the set of rules is fully-bounded. It has been shown in [6] that fully-bounded concurrent-Horn CTR is data complete for NP.

6 Other Uses of Logic in Workflow Modeling

Logic can be used is many different ways, and the approaches surveyed here are by no means the only ones where logic has been gainfully employed. One of the earliest works in this area is ACTA [10], which presented a methodology for specifying properties of complex transactions as axioms in the regular first-order predicate calculus. In particular, workflow dependencies of the kind discussed in Section 2 can be represented in this way. Thus, ACTA can be viewed as a workflow modeling framework. However, ACTA is not a complete framework. Because it is so general, it does not come with any special technique for verifying the properties of workflows and instead relies on a general-purpose theorem prover. Likewise, the ACTA framework does not come with a scheduler that could be used to enact workflows.

Vortex [18] uses logic in a different way. By itself Vortex is not based on a logic. However, declarative abstractions can be *derived* from Vortex workflows. For instance, one such abstraction models temporal dependencies among the invocations of activities in these workflows. This abstraction as well as workflow properties can then be represented as formulas in temporal logic and standard model-checking techniques can be used to verify these properties. Vortex also provides a rich language for specifying conditions on when different activities can be executed. However, in general, the scheduling problem for Vortex workflows is undecidable. It is reported that scheduling algorithms can be developed by imposing various restrictions on the form of the workflows [24].

An action logic based approach to workflow modeling is described in [5]. Workflows are specified as sets of triggers of the form "on $event_1$ if *condition* then $event_2$." In action logic, such triggers are represented as logical formulas with a well-defined semantics. Since triggers can be viewed as global intertask dependencies, this framework resembles the three frameworks discussed in the main part of this survey [4,29,13]. Correctness of a workflow is specified using formulas of the form "ϕ must hold after event sequence P at initial state σ." The approach comes with an algorithm for scheduling workflows to guarantee that the given correctness conditions hold. However, this algorithm is exponential in the size of the workflow. The constraints defined by the triggers are not as general as the ones in [4,29,13] in some respects, but on the other hand they have features that are missing in the other logic-based approaches (save [13]). For instance, triggers cannot define dependencies such as "if a happens then either b or c should also happen" or "if a then b must precede c." On the other hand, triggers allow conditions to be imposed on the state of the execution, which is not allowed in [4,29]. Although such conditions can easily be expressed in the CTR-based framework of [13], the algorithm developed there does not guarantee linear-time runtime scheduling under these circumstances.

7 Conclusions

In this paper we surveyed three approaches to modeling and managing workflows: one based on Temporal Logic [4,21], one on Event Algebra [29], and one on Concurrent Transaction Logic (CTR) [13,6]. At the moment, the approach based on CTR seems to be the most promising. Not only can it handle very general constraints, but it can also represent control flow graphs with transition conditions on the arcs. Apart from modeling, the CTR-based framework can also be used to check workflow specifications for consistency and for property verification. Together with the recent works on scheduling under resource allocation constraints and in non-cooperative environments [28,14], the CTR framework is clearly the most developed logic-based approach to workflow modeling.

Despite these early successes, the existing formal methods are not ready for prime time yet. Among the unsolved problems we can list exception handling (including recovery with compensation a la Sagas [19]),[7] scheduling of workflows with repeated events (*e.g.*, workflows with loops), and dynamic modification of workflows.

In addition, the emerging field of Web services [30,12] brings in the issues of modeling service ontologies, matchmaking and brokering algorithms, dialogs, and negotiation for services establishment. Web services standards provide languages for specifying technical, business-related, and process-related information about services. Such descriptions should enable two basic functions in the Web services model. First, they should allow applications to query and find services that satisfy their business requirements. Second, they should permit dynamic composition of service components and organize them into workflows.

These developments open up a vast area for research in formal methods for workflow modeling and verification with a real possibility of practical impact. In particular, we have identified the following challenging problems that could use input from the research community: (1) automatic identification of component services that match client's business rules, (2) behavioral analysis for determining how disparate services could be composed (perhaps in an optimal way with respect to some cost constraints) into a workflow that satisfies user's request, (3) coordination of multiple services from different, not necessarily trusted, participants, and (4) re-configuration of component services as client's requirements and operating conditions change.

There is a vast number of products on the market that claim to be workflow-related, but not all of them are WfMSs in the true sense of this word. For instance, IBM's Lotus Notes is often referred to as a workflow management system, but it really is just a groupware collaborative software. A long list of workflow-related products can be found in [33]. Commercial workflow tools, such as IBM's MQSeries Workflow, Oracle Workflow, and Fujitsu's iFlow provide support for business process modeling via flow charts for processes with static structure or via event-condition-action (ECA) type triggers for evolving and less structured processes. Currently, these solutions provide little or no support for querying repositories of existing processes to identify re-usable processes, and for composing them under constraints. Neither do they provide languages suitable for expressing workflow dependencies

[7] Note that ACTA, described in Section 6, can model Sagas-style compensation-based rollback. However, as we mentioned, ACTA is not a complete framework in many other ways.

beyond the very simple ones. We believe that the techniques surveyed here can be used to enrich the existing products. Moreover, as Web services become more and more popular, complex processes will be routinely constructed on ad hoc basis by a variety of users. In this environment, there will be growing need for logic-based formalisms and verification techniques that can ensure that workflow designs comply with their specifications.

Acknowledgements. We would like to thank the anonymous referees for their thorough reviews. This work was sponsored in part by the NSF grants IIS-0072927 and INT-9809945.

References

1. N.R. Adam, V. Atluri, and W.-K. Huang. Modeling and analysis of workflows using petri nets. In *Journal of Intelligent Information Systems 10(2)*, 1998.
2. G. Alonso, D. Agrawal, A. El Abbadi, and C. Mohan. Functionality and limitations of current workflow management systems. In *IEEE-Expert. Special issue on Cooperative Information Systems*, 1997.
3. A. Ankolenkar, M. Burstein, J.R. Hobbs, O. Lassila, S.A. McIlraith D.L. Martin, S. Narayanan, M. Paolucci, T. Payne, K. Sycara, and H. Zeng. Daml-s: A semantic markup language for web services. In *Intl. Semantic Web Working Symposium (SWWS)*, July 2001. http://www.semanticweb.org/SWWS/program/full/paper2.pdf.
4. P. Attie, M.P. Singh, A.P. Sheth, and M. Rusinkiewicz. Specifying and enforcing intertask dependencies. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1993.
5. C. Baral, J. Lobo, and G. Trajcevski. Formalizing workflows as collections of condition-action rules. In *Proceedings of the International Conference on Cooperative Information Systems (CoopIS)*, 1996.
6. A.J. Bonner. Workflows, transactions and datalog. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, 1999.
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. R. Chinnici, M. Gudgin, J.-J. Moreau, and S. Weerawarana. Web services description language (wsdl) version 1.2. Technical report, W3C, July 2002.
10. P.K. Chrysanthis and K. Ramamritham. ACTA: The SAGA continues. In *Transaction Models for Advanced Database Applications.* Morgan Kaufman, 1992.
11. F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. Business process execution language for web services, version 1.0. Technical report, IBM, July 2002.
12. F. Curbera, W.A. Nagy, and S. Weerawarana. Web services: Why and how. In *OOPSLA 2001 Workshop on Object-Oriented Web Services.* ACM, 2001.

13. H. Davulcu, M. Kifer, C.R. Ramakrishnan, and I.V. Ramakrishnan. Logic based modeling and analysis of workflows. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, 1998.
14. H. Davulcu, M. Kifer, and I.V. Ramakrishnan. A game logic for modeling and coordinating multi-agent workflows. in preparation, 2002.
15. U. Dayal, M. Hsu, and R. Ladin. Organizing long running activities with triggers and transactions. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 1990.
16. U. Dayal, M. Hsu, and R. Ladin. Organizing long-running activities with triggers and transactions. In *ACM SIGMOD Conference on Management of Data*, 1990.
17. E.A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, vol B*, 1990.
18. X. Fu, T. Bultan, R. Hull, and J. Su. Verification of vortex workflows. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2001.
19. H. Garcia-Molina and K. Salem. Sagas. In *Intl. Conference on Very Large Data Bases*, pages 249–259, May 1987.
20. 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.
21. R. Gunthor. Extended transaction processing based on dependency rules. In *Proceedings of International Workshop on Research Issues in Data Engineering*, 1993.
22. M. Huhns and M. Singh, editors. *Readings in Agents.* Morgan Kaufmann, San Francisco, CA, 1998.
23. R. Hull, F. Llirbat, E. Simon, J. Su, G. Dong, B. Kumar, and G. Zhou. Declarative workflows that support easy modification and dynamic browsing. In *Proceedings of the International Joint Conference on Work Activities Coordination and Collaboration*, 1999.
24. R. Hull, F. Llirbat, J. Su, G. Dong, B. Kumar, and G. Zhou. Adaptive execution of workflow: Analysis and optimization. In *Tech. Report, Bell Labs*, 1999.
25. J. Klein. Advanced rule-driven transaction management. In *IEEE COMPCON*. IEEE, 1991.
26. F. Leymann. Web services flow language (wsfl 1.0). Technical report, IBM, May 2001. http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf.
27. B. McKee, D. Ehnbuske, and D. Rogers. UDDI Version 2.0 API Specification. Technical report, UDDI.org, June 2001. http://www.uddi.org/.
28. P. Senkul, M. Kifer, and I.H. Toroslu. A logical framework for scheduling workflows under resource allocation constraints. In *Intl. Conference on Very Large Data Bases*, August 2002.
29. M.P. Singh. Semantical considerations on workflows: An algebra for intertask dependencies. In *Proceedings of the International Workshop on Database Programming Languages*, 1995.
30. T. Sollazzo, S. Handschuh, S. Staab, and M. Frank. Semantic web service architecture — evolving web service standards toward the semantic web. URL: http://citeseer.nj.nec.com/461405.html.
31. V.S. Subrahmanian, P. Bonatti, J. Dix, T. Eiter, and F. Ozcan. *Heterogeneous Agent Systems.* The MIT Press, Cambridge, MA, 2000.

32. A.H.M. ter Hofstede, M.E. Orlowska, and J. Rajapakse. Verification problems in conceptual workflow specifications. In *Proceedings of the International Conference on Conceptual Modeling*, 1996.
33. In *http://dmoz.org/Computers/Software/Workflow/Products/*.
34. W.M.P. van der Aalst. Verification of workflow nets. In *Application and Theory of Petri Nets*, 1997.
35. W.M.P. van der Aalst. The application of petri nets to workflow management. In *The Journal of Circuits, Systems and Computers 1(8)*, 1998.
36. D. Wodtke and G. Weikum. A formal foundation for distributed workflow execution based on state charts. In *Proceedings of the International Conference on Database Theory*, 1997.