Distrib. Syst. Engng 3 (1996) 222–238. Printed in the UK

Scheduling workflows by enforcing intertask dependencies*

P C Attie†+, M P Singh‡#, E A Emerson§††, A Sheth‖‡‡ and M Rusinkiewicz¶§§

† School of Computer Science, Florida International University, Miami, Florida 33199, USA
‡ Department of Computer Science, North Carolina State University, Raleigh, North Carolina 27695-8206, USA
§ Department of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712, USA
‖ Department of Computer Science, University of Georgia, Athens, Georgia, USA
¶ Department of Computer Science, University of Houston, Houston, Texas, USA

Abstract. *Workflows* are composite activities that can be used to support and automate multisystem applications involving humans, heterogeneous databases and legacy systems. The traditional atomic transaction model, successful for centralized and homogeneous applications, is not suitable for supporting such workflows. *Intertask dependencies*, which are conditions involving events and dependencies among workflow tasks, are used to specify the coordination requirements among the workflow tasks and are a central component of most workflow models. They form a basis for developing a uniform formal framework for workflows, which is a key contribution of this work. In this paper, we formalize intertask dependencies using temporal logic. This involves event attributes, which are needed to determine whether a dependency is enforceable and to properly schedule events. Each dependency is represented internally as a finite state automaton that captures the computations that satisfy the given dependency. Sets of automata are combined into a *scheduler* that produces global computations satisfying all relevant dependencies, thus enacting the given workflow. This algorithm is rigorously proved correct: it has been implemented.

1. Introduction

One of the main challenges in current database research is to design approaches—theories, languages, methodologies—for the development of applications that depend on related information stored in heterogeneous, multiple existing systems. The Carnot project at MCC addressed these challenges to build and support complex applications by developing tools and techniques [5]. *Relaxed task management* and *workflows* were key areas of work in this context [18, 31]. The architecture and results of this project are outlined in general terms in [33].

A workflow involves multiple (possibly heterogeneous) tasks (also called activities or steps) that need to be orchestrated or coordinated. A *task* is a computation that performs some useful function—database transactions are tasks of particular interest. To efficiently develop multisystem applications that involve humans and access existing heterogeneous and closed systems, we must be able to modularly capture the execution constraints of various applications. In many such systems, the data can be accessed only through the existing interfaces, even if they are internally stored under the control of a general-purpose DBMS. Such systems are frequently referred to as *legacy systems*. Complex applications that access several legacy systems or involve other forms of heterogeneity are better characterized as workflows than transactions. Workflows consist of related tasks executed on different systems with domain-specific semantic constraints among them. Because of their importance in modern, open, information environments, workflows are garnering much research interest.

The requirements of the traditional, so-called ACID, transaction model include full isolation and atomicity [20]. On the one hand, these translate into atomic commitment and global serializability, which can be too strong for multisystem applications. This is because they require the component systems to expose their internal states and wait for each other to progress far enough until all components can commit or abort in agreement. It is infeasible to modify existing legacy code (which is often undocumented) to provide synchronization and access to internal state. Such modification would also violate autonomy requirements of existing systems. On the other hand, it is crucial to

* This is a revised and extended version of [2].

+ E-mail address: attie@fiu.edu
‡ E-mail address: singh@ncsu.edu
†† E-mail address: emerson@cs.utexas.edu
‡‡ E-mail address: amit@cs.uga.edu
§§ E-mail address: marek@uh.edu

0267-1484/96/040222+17$19.50 © 1996 The British Computer Society, The Institution of Electrical Engineers and IOP Publishing Ltd

have some support for complex applications, so we cannot do away with all database primitives. For example, an application may need to ensure that two tasks commit only in a certain temporal order. An example is a banking application in which deposits made into an account over a certain period may have to be processed *before* debits are made from the account over the same period.

Therefore, we may need to selectively relax the ACID properties for multisystem transactions to capture precisely the synchrony and coupling requirements based on the true application semantics. While relaxed- or advanced-transaction models [10, 11, 24, 28, 29] have been proposed, there is continuing debate whether they are appropriate to support multisystem workflow applications [26, 34]. In any case, *intertask dependencies* provide a generic means to capture the constraints among tasks in a relaxed manner, independent of whether we follow a transaction model and, if so, which one. Intertask dependencies (also called task relationships or control dependencies), are constraints over *significant task events*, such as commit and abort. The concomitant reduction in constraints across tasks enables the generation of scripts that can be efficiently executed with a high level of parallelism. This, in turn, may result in higher availability of data, better response times, and higher throughput.

To illustrate these concepts, let us consider the following scenario. A travel agency maintains two databases: one containing detailed information about the bookings made by different agents and another containing a summary of the information in the first database with the number of bookings per agent. When the summary changes, a task is run that turns on an alarm if the summary falls below a preset threshold, and turns off the alarm if the summary rises above the threshold. An obvious integrity constraint is that for each travel agent, the number of rows in the bookings database should be equal to the number of bookings stored for that agent in the summary database.

If it holds initially, this constraint can be assured by executing all the updates to both databases as components of an atomic multidatabase transaction, which is globally serialized with other transactions [4]. This, however, is impossible if the database interfaces do not provide visible two-phase commit facilities. And, if they provide a visible precommit state, it would still be inefficient, requiring resources to be locked at remote sites. Instead, we assume that the interdatabase integrity is maintained by executing a workflow consisting of separate tasks that obey the appropriate intertask dependencies. These dependencies capture the requirement that if a delete task on the bookings database commits, then the corresponding decrement-summary task should also commit. Further, if a delete task aborts, while its associated decrement-summary task commits, then we must restore consistency by compensating for the spurious decrement. We do this by executing an increment-summary task. Figure 1 shows the tasks involved in this example; *dB*, *dS*, *iS* and *u?a* denote the delete booking, decrement summary, increment summary and update alarm tasks, respectively.

When the semantic constraints are relaxed as described above, the delete booking application becomes a workflow.

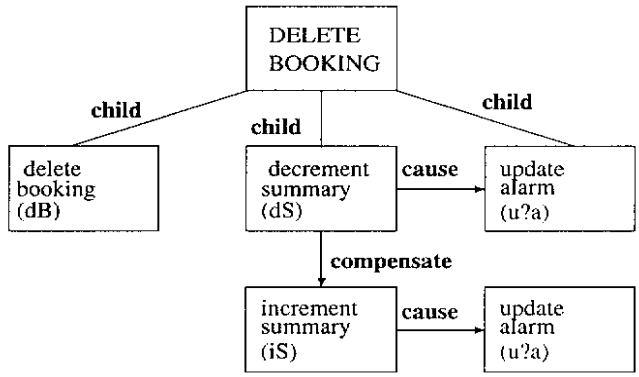


Figure 1. Task graph for the delete booking application.

This workflow will often leave the state of the combined database system inconsistent. However, it will ensure that consistency will eventually be restored. This can be justified on semantic grounds, since we know that temporary inconsistency is not damaging in this particular application. It might appear that we are introducing a requirement to reason about the semantics. However, the semantics is involved anyway, when programmers design complex applications. Our approach does not require any additional semantic knowledge. It simply provides high-level primitives to specify dependencies and thus declaratively capture what would otherwise be captured procedurally.

We model each intertask dependency as a *dependency automaton*, which is a finite state automaton whose paths represent the computations that satisfy the dependency. Each such automaton ensures that its corresponding dependency is not violated, by permitting only those events whose execution would not lead to the violation of the dependency. The *scheduler* receives events corresponding to a possible task execution. It queries the applicable dependency automata to determine whether they all allow the event to be executed. If so, the event is executed; otherwise, it is delayed (if delayable) and reattempted later.

We present a framework in which dependencies can be stated modularly as constraints across tasks. We also present a *scheduler* that *enforces* all stated dependencies, provided they are jointly enforceable, and assures that a dynamically changing collection of tasks is executed in accordance with the dependencies. It does this by appropriately accepting, rejecting, or delaying significant events. Our contributions include task behaviour modelling, formal modelling of intertask dependencies, and their provably correct enforcement. Additional details of workflow modelling that are consistent with this paper are discussed in [25]. The important issue of how dependencies are generated is complementary to the goals of this paper—see, e.g., [28, 30].

The rest of the paper is organized as follows. Section 2 provides the technical and methodological background for our work and illustrates it using the delete bookings application discussed above. Section 3 describes how we formally specify dependencies using the temporal logic CTL [13, 14], discusses event attributes and their impact on the enforceability of dependencies, and considers

how dependencies can be added or removed at run-time. Section 4 gives a formal definition of a dependency automaton, which we use to represent each dependency; it also shows how dependency automata operate and enforce their corresponding dependencies. Section 5 presents our execution model as well as the notion of *viable pathsets*, which we use as a correctness criterion. It formalizes these definitions and uses them in the definition of a scheduling algorithm. It also shows how a relaxed transaction model such as the Sagas [17] can be described (and hence enforced) as a set of dependencies. Section 6 briefly discusses the concurrency control, safety and recovery issues. A description of CTL and a rigorous proof of correctness of the scheduling algorithm are given in the appendices.

2. Background

The specification and enforcement of intertask dependencies has recently received much attention [6, 10, 11, 23, 25]. Following [6], we specify intertask dependencies as constraints on the occurrence and temporal order of certain significant events. Klein has proposed the following two primitives [23]:

(i) $e_1 \rightarrow e_2$: If e_1 occurs, then e_2 must also occur. There is no implied ordering on the occurrences of e_1 and e_2.
(ii) $e_1 < e_2$: If e_1 and e_2 both occur, then e_1 must precede e_2.

Well known examples of dependencies include:

- Commit dependency [6]: Transaction A is commit-dependent on transaction B, iff if both transactions commit, then A commits before B commits. Let the relevant significant events be denoted as cm_A and cm_B. This can be expressed as $cm_A < cm_B$.
- Abort dependency [6]: transaction A is abort-dependent on transaction B, iff if B aborts, then A must also abort. Let the significant events here be ab_A and ab_B, so this can be written $ab_B \rightarrow ab_A$.
- Conditional existence dependency [23]: if event e_1 occurs, then if event e_2 also occurs, then event e_3 must occur. That is, the existence dependency between e_2 and e_3 comes into force if e_1 occurs. This can be written $e_1 \rightarrow (e_2 \rightarrow e_3)$.

Note that we allow dependencies of the form $E_1 \rightarrow E_2$, where E_1 and E_2 are general expressions. An expression E can be formally treated as an event by identifying it with the first event occurrence that makes it definitely true. For example, $e_2 \rightarrow e_3$ is made true as soon as e_3 or the complement of e_2 occurs, the complement of e_2 being an event whose occurrence implies that e_2 will never occur.

The relationships between the significant events of a task can be represented by a state transition diagram, which serves as an abstraction for the actual task by hiding irrelevant details of its internal computations. This is akin to the intuition behind the synchronization skeletons of [14]. The execution of an event causes a transition of the task to another state. Figure 2 shows an example task state transition diagram, based on figure 17.4 of [15, p 535].

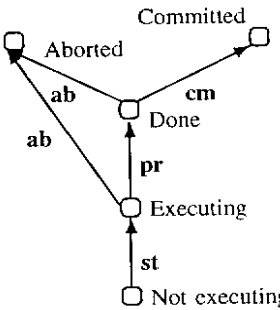
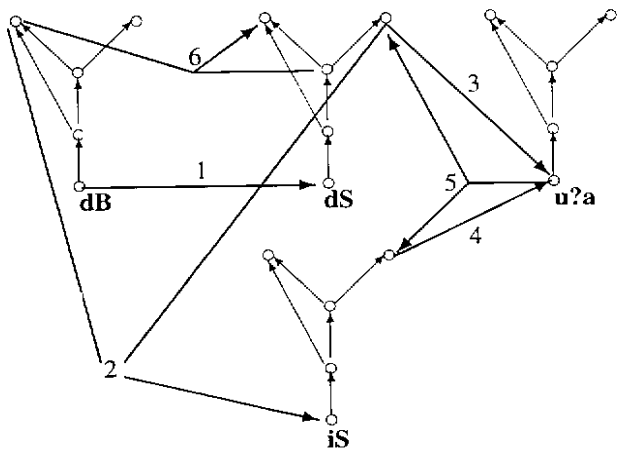


Figure 2. An example task state transition diagram.

From its initial state (at the bottom of the diagram), the task first executes a start event (**st**). Once the task has started, it will eventually either abort, as represented by the **ab** transition, or finish, as represented by the **pr** transition (for 'done and prepared to commit'). When a task is done, it can either commit, i.e., make the **cm** transition, or abort, i.e., make the **ab** transition. For simplicity, we omit the 'forget' transition, which comes after **ab** and **cm**.



1.	$st(dB) \rightarrow st(dS)$
2.	$ab(dB) \rightarrow (cm(dS) \rightarrow st(iS))$
3.	$cm(dS) \rightarrow st(u?a)$
4.	$cm(iS) \rightarrow st(u?a)$
5.	$st(u?a) \rightarrow (cm(dS)$ or $cm(iS))$
6.	$(ab(dB)\ \&\ (ab(dB) < pr(dS))) \rightarrow ab(dS)$

Figure 3. Intertask dependencies in the delete booking example.

Using the state transition diagrams and significant events defined above, we can represent the travel agent workflow described in the previous section as shown in figure 3. The intertask dependencies are shown as 'links' between states that result after the corresponding significant events of the different tasks are performed (& denotes conjunction). For simplicity, we consider only the dependencies shown in figure 3. These dependencies mean (1) start dS when dB begins; (2) compensate spurious dS by executing iS; (3) start $u?a$ when summary changes; (4) start $u?a$ when summary changes; (5) start $u?a$ only when summary changes; and (6) abort dS if dB aborts, unless dS has already committed.

3. Intertask dependency declarations

As discussed in section 2, we specify intertask dependencies as constraints on the occurrence and temporal order of events. The significant events and transitions of a task depend on the characteristics of the local system where it executes. Our theory and implementation apply to tasks with an arbitrary set of task states and significant events. We assume that an event can occur at most once in any possible execution of the system. This is not a restriction in real terms. If a task aborts and must be reexecuted, a new id may be generated for it (and for its events). The dependencies can be appropriately modified and everything can proceed normally.

3.1. Formal specification of dependencies

We adopt the language of Computation Tree Logic (CTL) as the language of our dependencies [13]. CTL is a powerful language, well known from distributed computing [3, 7, 9, 13, 14]. A brief description of CTL and modelling of various dependencies is given in appendix A. The primitives $<$ and $\rightarrow$ are useful macros that yield CTL formulae. CTL can uniformly express different dependencies. Since it is a formal language, it helps reduce ambiguity in communication. It also makes it possible to formally determine the relationships among different dependencies, e.g., whether they are consistent, or whether one entails another.

The dependencies should be easily specifiable by users or database administrators. For this reason, it is essential that the automata that enforce those dependencies be synthesized automatically from those dependencies. CTL formulae can be used to automatically synthesize dependency automata: this process is hidden from the dependency specifier. Thus we retain the flexibility of Klein's approach, while using a formal, more expressive and general representation for which a significant body of research already exists.

3.2. Enforceable dependencies

The scheduler enforces a dependency by variously allowing, delaying, rejecting or forcing events to occur, so that the resulting computation satisfies the given dependency. Some syntactically well-formed dependencies may not be enforceable at run-time. For example, the dependency $ab(T_1) \rightarrow cm(T_2)$ is not enforceable, because a scheduler can neither prevent $ab(T_1)$ from occurring nor in general guarantee the occurrence of $cm(T_2)$. This is because, in general, a scheduler cannot prevent tasks from unilaterally deciding to abort. Thus both T_1 and T_2 can abort.

We associate the following attributes with significant events that meet the given conditions:

- forcible, whose execution can be forced;
- rejectable, whose execution can be prevented;
- delayable, whose execution can be delayed.

We assume below that local systems on which the tasks are executed provide a prepared-to-commit state so that a task can issue a *prepare-to-commit* (**pr**) event. The prepared-to-commit state is *visible* if the scheduler can decide whether the prepared task should commit or abort. Table 1 shows the attributes of the significant events of transactions commonly found in database applications and DBMSs. Therein, a $\surd$ indicates that the given attribute always holds, whereas a $\times$ indicates that the given attribute may or may not hold.

Table 1. Attribute tables for *some* common significant events.

Event	Forcible?	Rejectable?	Delayable?
cm	$\times$	$\surd$	$\surd$
ab	$\surd$	$\times$	$\times$
pr	$\times$	$\times$	$\times$
st	$\surd$	$\surd$	$\surd$

Let e, e_i, e_j, etc denote any significant event and $D(e_1, \ldots, e_n)$ denote an unspecified dependency over $e_1, \ldots, e_n$. We can characterize the enforceability of dependency $D(e_1, \ldots, e_n)$ in terms of the attributes of $e_1, \ldots, e_n$. For example, $e_1 \rightarrow e_2$ is run-time enforceable if rejectable(e_1) and delayable(e_1) hold, since we can then delay e_1 until e_2 is submitted, and reject e_1 if we see that the task that issues e_2 has terminated (or timed out: see below) without issuing e_2. Alternatively, if e_2 is forcible, then we can enforce $e_1 \rightarrow e_2$ at run-time by forcing the execution of e_2 when e_1 is accepted for execution. Yet another (although somewhat vacuous) strategy would be to unconditionally reject e_1. This strategy is available if rejectable(e_1) holds.

As another example, consider $e_1 < e_2$, for which there are two possible strategies. The first, which can be applied if delayable(e_2) holds, is to delay e_2 until either e_1 has been accepted for execution, or task 1 has terminated without issuing e_1. The second, which can be applied if rejectable(e_1) holds, is to let e_2 be executed when it is submitted and thereafter reject e_1 if it is submitted.

One way to extend our approach to real-time dependencies is by considering real-time events, such as clock times (e.g., 5:00 p.m.), as regular events that lack the attribute of delayability. Consider $e_1 <$ 5:00 p.m.. This dependency is enforceable only if e_1 is rejectable. The scheduler can enforce $e_1 <$ 5:00 p.m. by accepting e_1 if 5:00 p.m. has not already occurred (i.e., if it is before 5:00 p.m.) and by rejecting e_1 otherwise.

3.3. Dynamic addition and removal of dependencies

The preceding exposition assumed that all dependencies are initially given, i.e., at compile-time. However, dependencies may be added or deleted dynamically at run-time. The removal of a dependency is achieved simply by removing its corresponding automaton. The addition of a dependency requires that an automaton be synthesized for it and used in further scheduling. A dependency may be added too late to be enforced. Suppose $D = e_1 \rightarrow e_2$ is added after e_1 occurs. If e_2 is not forcible and is never

submitted, D cannot be enforced. This is unavoidable in general, since the addition of dependencies cannot be predicted. At best we can report a violation when such a dependency is added.

4. Dependency automata: enforcing a single dependency

For each dependency D, we assume a finite state machine A_D that is responsible for enforcing D. A_D captures all possible orders of events for which D is satisfied. The results of this paper are orthogonal to how the automata are synthesized. For well known dependencies, it is most effective to take the automata from a precompiled library of automata. We return to this topic in section 7.

A_D is a tuple $\langle s_0, S, \Sigma, \rho \rangle$, where S is a set of states, s_0 is the distinguished initial state, Σ is the alphabet, and $\rho \subseteq S \times \Sigma \times S$ is the transition relation. We use t_i to indicate the specific termination event of task i, and ε to denote any event which can either be a significant event (denoted with e) or a termination event. We discuss the generation and usage of termination events below. The elements of Σ are denoted as σ, σ', etc. σ can be of any of the forms described below.

- $a(\varepsilon_1, \ldots, \varepsilon_m)$: this indicates that A_D accepts the events ε_1 through ε_m. If this transition is taken by A_D, then each ε_i is accepted and, if ε_i is a significant event, it is then forwarded to the event dispatcher for execution.
- $r(e_1, \ldots, e_m)$: this indicates that A_D rejects the events e_1 through e_m because the execution of any of them would violate the dependency D.
- $\sigma_1 ||| \ldots ||| \sigma_n$, where the $\sigma_i \in \Sigma$: this indicates the interleaving of the accept operations corresponding to σ_1 through σ_n.
- $\sigma_1; \ldots; \sigma_n$, where the $\sigma_i \in \Sigma$: this indicates the accept operations of σ_i occur before the accept operations of σ_{i+1} (for $1 \leq i \leq (n-1)$).

Note that reject operations do not result in actual operations being executed by the dispatcher, but merely cause the sending of a reject message to each task that submitted a rejected event.

Example dependency automata

We represent A_D as a labelled graph, whose nodes are states, and whose edges are transitions. Each edge is labelled with an element σ of Σ. σ denotes the actions, such as accept or reject, that are taken by the scheduler when that transition is executed.

In figures 4 and 5, we give example dependency automata for the dependencies $e_1 < e_2$, and $e_1 \rightarrow e_2$, respectively. The symbol | indicates choice: an edge labelled $\sigma|\sigma'$ may be followed if the scheduler permits either σ or σ'.

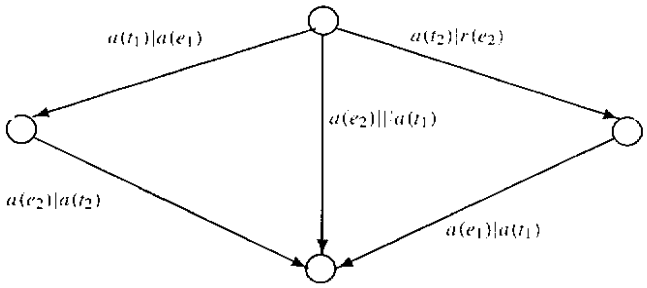


Figure 4. Dependency automaton for order dependency $e_1 < e_2$ assuming rejectable(e_2) and delayable(e_2).

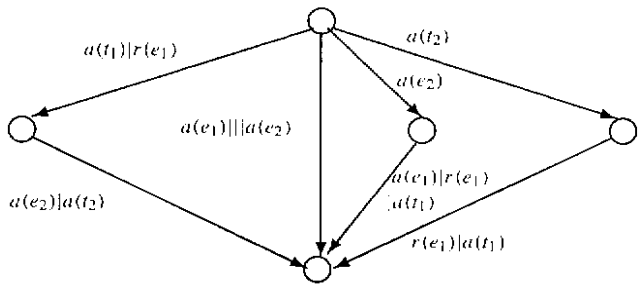


Figure 5. Dependency automaton for existence dependency $e_1 \rightarrow e_2$ assuming rejectable(e_1) and delayable(e_1).

The operation of an automaton

The automaton A_D operates as follows. At any time, it is in some state, say, s. Initially, $s = s_0$. Events arrive sequentially. Let ε be the current event. If s has an outgoing edge labelled $a(\varepsilon)$ and incident on state s', then the given transition is enabled. This means that, as far as its local state is concerned, A_D can change its state to s'. However, A_D cannot actually make the transition unless the scheduler permits it (see section 5).

If the scheduler permits a certain transition, then the automaton can execute it, thereby changing its local state to keep in synchronization with respect to the events executed so far. The behaviour of the scheduler is such that it accepts an event only if it can find an event ordering that is consistent with all of the dependency automata that contain that event in their alphabet. So if it accepts an event, all the relevant automata must be in agreement. Therefore, each of them must execute the given accepting transition. This ensures that acceptance of the event does not violate any of the dependencies in which the event is mentioned. Similarly, the scheduler can reject an event only if all of the relevant automata reject it, i.e., only if it can find an event ordering that is consistent with all of the relevant dependency automata executing a rejecting transition for the event. The same reasoning as for accepting an event applies here, since the rejection of an event can also cause the violation of a dependency in which the event is mentioned Section 5 discusses the operation of the scheduler in detail.

The following observations concern how a dependency automaton enforces a dependency. A t_i indicates the termination or timing out of task i. A dependency automaton cannot reject a t_i event, since it cannot unilaterally prevent such an event. The importance of t_i events is that their submission tells the automaton that

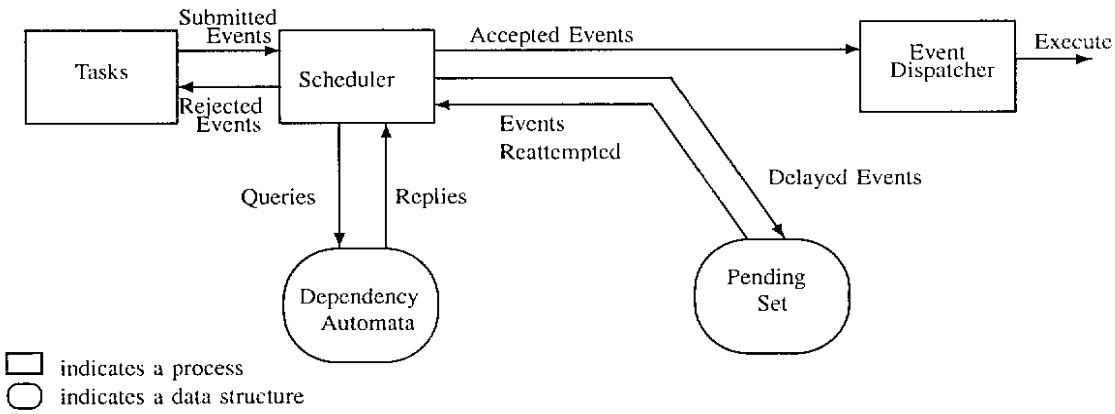


Figure 6. The execution model.

events that may have been submitted by the given task will definitely not be submitted. This can significantly affect the automaton's behaviour. Knowledge that the given task has terminated may allow the scheduler to accept for execution a previously delayed event e_j, as the knowledge that e_i will never occur may enable the scheduler to infer that the execution of e_j now will not violate certain dependencies that it might have violated before. This happens, for example, if a dependency $e_i < e_j$ is to be enforced and e_j has been submitted, but is being delayed. In such a case, the arrival of t_i ensures that the dependency $e_i < e_j$ cannot be violated; consequently, e_j can be scheduled (unless doing so would violate some other dependencies).

Dealing with failures using timeouts

We have so far interpreted the t_i events to indicate the termination of task i. Ordinarily, tasks terminate by committing or aborting. However, system problems, such as disk crashes and communication failures, may cause indefinite waits. For example, the automaton for $e_1 < e_2$, shown in figure 4, delays accepting e_2 until t_1 or e_1 is submitted. Thus, this automaton could possibly hang forever, if neither t_1 nor e_1 is forthcoming.

One policy is to have the automaton accept e_2 when e_2 arrives and reject e_1 if e_1 arrives later. In general, this policy speeds up e_2's task at the cost of aborting e_1's task and, possibly, delaying or aborting the global task. In cases where both policies, namely one in which an event is indefinitely delayed and the other in which an event is eagerly rejected, are unacceptable, a policy based on timeouts may be preferred. This would require tasks to wait, but would allow timeouts to be generated when expected events are not received within a reasonable time. This is an improvement in practical terms, but does not require any significant change in our approach. We support timeouts by modifying the interpretation of the t_i events in the above and associate them with either the normal termination of a task or a timeout on the corresponding event, e_i. We assume that e_i is not submitted after t_i has been submitted.

5. The scheduler: enforcing multiple dependencies

A system must enforce several dependencies at the same time. A naive approach would generate a product of the individual automata (A_Ds) that each enforce a single dependency. However, if there are m individual automata each roughly of size N, then the product automaton has size of the order of N^m. This is intractable for all but the smallest m. We avoid this 'state explosion problem' [8], by coordinating the relevant individual automata at run-time rather than building a static (and exponentially large) product at compile-time, using techniques similar to those of [3]. Although the worst case time complexity is still exponential, we have reason to believe that in many interesting cases, e.g., certain workflows in telecommunications applications [1], the time complexity is polynomial. Also, the space complexity of our technique is polynomial versus the exponential space complexity of building the product automaton

5.1. The execution model

Figure 6 shows the execution model. Events are submitted to the scheduler as tasks execute. We introduce the correctness criterion of *viable pathsets*, which is used to check whether all dependencies can be satisfied if a given event is executed. Computing a viable pathset requires looking at all relevant dependency automata. If an event can be accepted based on the viable pathset criterion, it is given to the event dispatcher for execution. If an event cannot be accepted immediately, then it still may be possible to execute it after other events occur, provided that the event is delayable. In that case, the event is put in the pending set and a decision taken on it later. If the scheduler ever permits the execution of an $r(e)$ transition by some automata, then e is rejected, and a $reject(e)$ message is sent to the task that submitted e to the scheduler.

5.2. Pathsets

We now discuss pathsets, present an algorithm to compute them, and discuss event execution in more detail. When an event ε is submitted, the scheduler searches for a

pathset, i.e., a set of paths with one path from each relevant dependency automaton. The desired pathset must

(i) accept ε;
(ii) begin in the current global state of the scheduler;
(iii) be order-consistent;
(iv) be *a-closed* and *r-closed*; and
(v) be executable.

A pathset accepts ε iff all its member paths mentioning ε should accept it and there should be no paths accepting the termination event associated with ε. Order-consistency means that different paths in the set must agree on the order of execution of each pair of events. The requirements of *a-closure* and *r-closure* mean that for any event that is accepted or rejected, paths from each automaton referring to that event must be included and must agree on whether to accept or reject it. Executable means that all rejected events must have been submitted and all accepted events must have been submitted or be forcible. A pathset that meets criteria (ii)–(v) is called *viable*. These conditions ensure that the events in the member paths of Π can be executed in the order imposed by those paths.

After some technical definitions, we give further intuitions and present an algorithm to compute pathsets.

Definition 1. (Global state). A *global state* s is a tuple $\langle s_{D_1}, \ldots, s_{D_i}, \ldots, s_{D_n} \rangle$ where s_{D_i} is the local state of A_{D_i}, and $D_1, \ldots, D_n$ are all the dependencies in the system. $s{\uparrow}D_i = s_{D_i}$ denotes the project of s onto D_i.

The global state is simply the aggregation of the local states of every individual dependency automaton.

Definition 2. (Path). A *path* π_D in A_D is a sequence $s^1 \xrightarrow{\sigma^1} s^2 \xrightarrow{\sigma^2} \ldots$ such that $(\forall j \geq 1 : (s^j, \sigma^j, s^{j+1}) \in \rho_D)$ where ρ_D is the transition relation of A_D.

A global computation is a sequence of events as executed by the event dispatcher. Recall that A_D is meant to encode all the computations that satisfy dependency D. Thus, each path of A_D represents computations that satisfy D. Further, A_D is maximal in the sense that every possible computation whose prefixes satisfy D is represented by some path in A_D. By definition, a global computation must consist solely of events accepted by the scheduler. Our scheduler has the property that, for each dependency D, the projection of any global computation onto the events in D is represented by a path in A_D. This means that our scheduler enforces each dependency.

Definition 3. (Pathset). A *pathset* Π is a set of paths such that:

(i) Each element of Π is a path in some A_D.
(ii) Each A_D contributes at most one path to Π.

Definition 4. (Source state). A *global state* s is a source state of Π iff for every π_D in Π: the first state of π_D is $s{\uparrow}D$.

A source state of a pathset is a global state in which execution of the pathset may commence.

Definition 5. (Depset). The *depset* of an event ε is the set of automata in which ε is some part of some label (i.e., ε is accepted or rejected).

The depset gives the set of automata that are potentially relevant for scheduling it.

Definition 6. (Accepts). A *pathset* Π accepts event ε iff for every A_D in $depset(\varepsilon)$: there exists a $\pi_D \in \Pi$ such that $a(\varepsilon)$ occurs along π_D.

Thus execution of Π entails execution of ε. The scheduler attempts to execute a submitted event ε by searching for a pathset that accepts ε.

Definition 7. (Rejects). A *pathset* Π rejects event ε iff for every A_D in $depset(\varepsilon)$: there exists a $\pi_D \in \Pi$ such that $r(\varepsilon)$ occurs along π_D.

Thus execution of Π entails rejection of ε. The agent that submitted ε is then informed of this rejection.

Definition 8. (a-closed). A *pathset* Π is a-closed iff for every $a(\varepsilon)$ that occurs along some π in Π: Π accepts ε.

An event ε is executed iff every automaton A_D such that $\varepsilon \in \Sigma_{A_D}$ executes an $a(\varepsilon)$ transition. Thus, if some path π in pathset Π contains an $a(\varepsilon)$ transition, then that transition can only be taken if an $a(\varepsilon)$ transition in every A_D such that $\varepsilon \in \Sigma_{A_D}$ is taken, i.e., if Π accepts ε.

Definition 9. (r-closed). A *pathset* Π is r-closed iff for every $r(\varepsilon)$ that occurs along some π in Π: Π rejects ε.

An event ε is rejected iff every automaton A_D such that $\varepsilon \in \Sigma_{A_D}$ executes an $r(\varepsilon)$ transition. Thus, if some path π in pathset Π contains an $r(\varepsilon)$ transition, then that transition can only be taken if an $r(\varepsilon)$ transition in every A_D such that $\varepsilon \in \Sigma_{A_D}$ is taken, i.e., if Π rejects ε.

Definition 10. (Mutually order-consistent). Paths π_1, π_2 are *mutually order-consistent* iff there do not exist events $\varepsilon_1, \varepsilon_2$ such that:

(i) $a(\varepsilon_1)$ and $a(\varepsilon_2)$ both occur along π_1, and $a(\varepsilon_1)$ occurs before $a(\varepsilon_2)$ in π_1.
(ii) $a(\varepsilon_1)$ and $a(\varepsilon_2)$ both occur along π_2, and $a(\varepsilon_2)$ occurs before $a(\varepsilon_1)$ in π_2.

This states that π_1 and π_2 do not impose opposite orderings on any pair of events, ε_1 and ε_2 to be executed. Thus it is possible to construct a global computation that has both π_1 and π_2 as local projections. Note that reject operations, e.g., $r(\varepsilon_1)$, $r(\varepsilon_2)$, do not need to be so ordered, since they do not contribute events to the global computation.

Definition 11. (Order-consistent). A pathset Π is *order-consistent* iff for all π_1 and π_2 in Π: π_1 and π_2 are mutually order-consistent.

This states that no pair of paths in pathset Π imposes opposite orderings on some pair of events. Thus it is possible to construct a global computation that has all the paths in Π as local projections.

Definition 12. (Executable). A pathset Π is *executable* iff for every $a(\varepsilon)$ or $r(\varepsilon)$ that occurs along some path π in Π: event ε has been submitted to the scheduler.

This reflects the constraint that neither an $a(\varepsilon)$ transition nor an $r(\varepsilon)$ transition can be executed unless ε has been submitted by its agent to the scheduler.

```
search_PS(AS_due, ES_a, ES_r, PS, ES_Done_a, ES_Done_r, Curr_St)

if (ES_a ∪ ES_r) = ∅ and AS_due = ∅ then
   return;
else {  /* Add to the automata set */
   AS_due := AS_due ∪ depset(ES_a ∪ ES_r);
   ES_Done_a := ES_Done_a ∪ ES_a;
   ES_Done_r := ES_Done_r ∪ ES_r;
   A := select_a(AS_due);        /* select an automaton */
   AS_due := AS_due − {A};
   P := get_candidate_paths(A, ES_Done_a ∩ Σ_A, ES_Done_r ∩ Σ_A, PS, Curr_St ↑ A);
   if A ∈ automata_in(PS) then PS := PS − {path_of(A, PS)};
   while P ≠ ∅
         p := select_p(A, P);
         /* check path p */
         P := P − {p};
         AS_due' := AS_due;
         ES_a' := a_events({p}) − ES_Done_a;
         ES_r' := r_events({p}) − ES_Done_r;
         PS' := PS ∪ {p};
         ES_Done_a' := ES_Done_a;
         ES_Done_r' := ES_Done_r;
         search_PS(AS_due', ES_a', ES_r', PS', ES_Done_a', ES_Done_r', Curr_St);
         if PS' ≠ ∅ then /* PS' is viable; end all recursive calls */
               AS_due := AS_due';
               ES_a := ES_a';
               ES_r := ES_r';
               PS := PS';
               ES_Done_a := ES_Done_a';
               ES_Done_r := ES_Done_r';
               return;
   endwhile
   /* all paths in P failed, so return ∅ */
   PS := ∅;
   return;
   }
```

Figure 7. Pathset search algorithm.

Definition 13. A pathset Π that satisfies the following conditions is called *viable*.

(i) Π has the current global state as a source state
(ii) Π is a-closed and r-closed
(iii) Π is order-consistent
(iv) Π is executable.

These conditions ensure that the events in the member paths of Π can be executed in the order imposed by those paths.

Now when an event ε is submitted to the scheduler, the scheduler attempts to execute ε by finding a viable pathset Π that accepts ε. If such a pathset is found, then all of the events that are accepted by the pathset are executed in an order that is consistent with that imposed by the pathset. This results in the global state of the scheduler being updated appropriately. If such a pathset is not found, then event ε is placed in the pending set. Another attempt at finding a suitable pathset may be made when more events that have an impact on the acceptability of ε have been submitted. Event ε remains in the pending set until either a viable pathset is found, or a viable pathset is executed that rejects ε. In the latter case, ε is rejected for execution by the scheduler and a message is sent to ε's agent informing it of this rejection.

5.3. The pathset search algorithm

Figure 7 presents a (recursive) procedure **search_PS** that searches for viable pathsets. search_PS is initially called as **search_PS**($\emptyset$, $\{\varepsilon\}$, $\emptyset$, $\emptyset$, $\emptyset$, $\emptyset$, **s**), where ε is an event that the scheduler is attempting to execute, and **s** is the current global state. It attempts to construct a viable pathset by selecting paths from each automaton in AS_due that are order-consistent with PS and executable. If these paths contain $a(\varepsilon)$ or $r(\varepsilon)$ events that occur in automata outside AS_due, it adds those automata to AS_due, thereby ensuring a-closure and r-closure of the eventual solution. Then automata in AS_due are examined one by one by invoking search_PS recursively. Each invocation searches through the paths of a single automaton (given by the select_a function) and either finds a solution or recurses. Note that the successful return of a viable pathset only occurs when AS_due, ES_a, and ES_r have each been reduced to the empty set, i.e., no edges are pending, and all the

```
get_candidate_paths(A, ES_a, ES_r, PS, s0)
  if A ∈ automata_in(PS) then
      p := path_of(A, PS);
      s := last(p);
      ES_a' := ES_a − a_events({p});
      ES_r' := ES_r − r_events({p});
      Done_a' := a_events({p});
      Done_r' := r_events({p});
      P' := get_cands(A, ES_a', ES_r', Done_a', Done_r', PS, s);
      return({pp' | p' ∈ P'}); /* juxtaposition denotes path concatenation */
  else
      return(get_cands(A, ES_a, ES_r, ∅, ∅, PS, s0));
  endif
```

Figure 8. Get candidate paths function.

necessary automata have been examined and an appropriate path selected from each. The arguments of **search_PS** are:

- AS_due: the automata that the given invocation must consider.
- ES_a: the events that the given invocation must accept.
- ES_r: the events that the given invocation must reject.
- PS: the pathset that has already been computed. The given invocation must extend this set, either by adding paths from automata that have not yet been considered, or by extending the paths already in the set.
- ES_Done_a: the events that must be accepted and for which the automata in their dependency sets have already been added to AS_Due.
- ES_Done_r: the events that must be rejected and for which the automata in their dependency sets have already been added to AS_Due.
- Curr_St: the current global state. Each of the paths that constitute the returned pathset must begin at a local state projected from this global state.

The data structures of search_PS are:

- A: the dependency automaton currently being searched for a path possibly to be added to the pathset being computed.
- P: the set of paths in A that are executable and order-consistent with respect to the current pathset. These paths are tested for suitability for inclusion in the pathset being computed.
- ES_a': the set of events that must be accepted and for which the current pathset is not a-closed.
- ES_r': the set of events that must be rejected and for which the current pathset is not r-closed.
- PS': the pathset returned by the next recursive invocation of search_PS.

search_PS also uses the following functions:

- a_events(PS): the set of events, ε, for which $a(\varepsilon)$ occurs along some path in PS.
- r_events(PS): the set of events, ε, for which $r(\varepsilon)$ occurs along some path in PS.
- depset(ε): as defined above in this section, depset(ε) = $\{A_D | \varepsilon \in \Sigma_D\}$.
- automata_in(PS): the set of dependency automata that contribute a path to PS.
- path_of(A, PS): the path contributed by automaton A to pathset PS.
- select_p(A, P): returns an arbitrary element of P.
- select_a(AS_due): returns an arbitrary element of A.

The implementation of the above functions and predicates is straightforward and is therefore omitted. The annotations of **search_PS** use the following predicates and functions:

- o_cons(PS): returns true iff PS is order-consistent.
- exec(PS): returns true iff PS is executable.
- automata_in(PS, γ): the set of dependency automata that contribute a path (along which γ occurs) to PS. γ is either $a(\varepsilon)$ or $r(\varepsilon)$ for some ε.

search_PS calls the function **get_candidate_paths(A, ES_a, ES_r, PS, s0)** (given in figure 8), which returns the set of paths in automaton A that are candidates for inclusion into the pathset PS that **search_PS** is building. **get_candidate_paths** calls the (recursive) function **get_cands(A, ES_a, ES_r, Done_a, Done_r, PS, source)** (given in figure 9) which searches recursively through A as it builds up the appropriate candidate paths. **get_candidate_paths** and **get_cands** use the following functions (as well as some of those listed above):

- last(p): returns the last state in path p.
- out_edges(s, A): returns the set of transitions in automaton A that have state s as a source state.

and the following predicates:

- order_consistent(ε, Done_a, Done_r, PS): returns true iff there is no path p in PS such that, along p, ε occurs before some event in Done_a ∪ Done_r.
- executable(ε): returns true iff ε has been submitted, or ε is a forcible event (in which case, the scheduler can itself initiate the execution of ε).

The symbol λ used in figure 9 denotes the empty path.

```
get_cands(A, ES_a, ES_r, Done_a, Done_r, PS, source)
  if ES_a = ∅ ∧ ES_r = ∅ then
    return(λ);
  else
    Selected_P := ∅;
    Edge_set := out_edges(source, A);
    for ε ∈ Edge_set do
        if executable(ε) and order_consistent(ε, Done_a, Done_r, PS) then
          dest := target_node(ε);
          P := get_cands(A, ES_a' − a_events({ε}), ES_r' − r_events({ε}),
                         Done_a ∪ a_events({ε}), Done_r ∪ r_events({ε}), PS, dest);
          Selected_P := Selected_P ∪ {ε;p | p ∈ P};
    endfor;
    return(Selected_P);
  endif
```

Figure 9. Generate candidate paths function.

5.4. The scheduler

The scheduler is a nonterminating loop, which on each iteration attempts to execute an event ε that has just been submitted or is in the pending set (figure 6). It does this by invoking search_PS($\emptyset$, $\{\varepsilon\}$, $\emptyset$, $\emptyset$, $\emptyset$, $\emptyset$, s). If this invocation returns a nonempty Π, then Π is immediately executed. Otherwise, ε is placed in the pending set. Π is executed by (a) accepting the events that Π accepts in a partial order that is consistent with Π and (b) rejecting all events rejected by Π.

Definition 14. (Path projection). The projection $\eta{\uparrow}D$ of global computation η onto a dependency automaton D is the path obtained from η by removing all transitions ε such that $\varepsilon \notin \Sigma_D$.

Lemma 1. Let η be a global computation generated by the scheduler. Then, for every dependency D, $\eta{\uparrow}D$ is a path in A_D.

Proof sketch. By construction of the scheduler. □

5.5. Complexity of the pathset search algorithm

Let $A_{D_1}, \ldots, A_{D_n}$ be all the dependency automata in the system. For any automaton A_{D_i} and any event ε in the alphabet of A_{D_i}, there will, in general be several local paths in A_{D_i} containing ε. Let ℓ denote the maximum number of such paths, taken over all events and automata in the system. Then, for any invocation of search_PS, there will be $O(n^\ell)$ candidate pathsets in the worst case. Since search_PS works by searching for a viable candidate pathset, it will have $O(n^\ell)$ time-complexity in the worst case. We note, however, that when $\ell = 1$, this complexity is linear in n.

Although assuming $\ell = 1$ is too restrictive in general, a typical situation in workflows is that a single stimulus from the 'environment' sets up a long chain of activity. More precisely, there are many dependencies of the form $e_1 \rightarrow e_2$ and $e_1 < e_2$ where e_2 is both delayable and forcible. In this situation, the dependency automata for $e_1 \rightarrow e_2$ and $e_1 < e_2$ both consist of a single path containing $a(e_1)$ followed by $a(e_2)$. Thus, we partition the automata $A_{D_1}, \ldots, A_{D_n}$ into two classes:

- *simple* automata, which contain exactly one local path for each event in their alphabet
- *complex* automata, which contain more than one local path for some event in their alphabet

Let the numbers of simple and complex automata in the system be ns, nc respectively (note $ns+nc = n$). The worst case time complexity of search_PS can now be rewritten as $O(ns + nc^\ell)$. In cases where $nc << n$, the exponential factor nc^ℓ will not be a hindrance to practical application.

5.6. Example of scheduler operation

We now give an example of how relaxed transactions expressed with $<$ and $\rightarrow$ can be scheduled using our algorithm. For simplicity, let the only dependencies in force be $e_1 < e_2$ and $e_1 \rightarrow e_2$, where both e_1 and e_2 are rejectable and delayable. Let $A_<$ and $A_\rightarrow$ be the corresponding automata as shown in figures 4 and 5. Assume that e_1 is submitted first. We find $a(e_1)$ in $A_<$. However, since no path in $A_\rightarrow$ begins with $a(e_1)$, the empty pathset is returned and e_1 added to the pending set. When e_2 is submitted, two executable paths can be found in $A_\rightarrow$: $a(e_2); a(e_1)$ and $a(e_2)|||a(e_1)$. The a-closure requirement now forces the scheduler to search $A_<$ for a path that accepts e_1 and e_2. The only such path is $a(e_1); a(e_2)$. Since $a(e_1); a(e_2)$ and $a(e_2); a(e_1)$ are not mutually order-consistent, the only viable pathset is $\{a(e_1); a(e_2), a(e_2)|||a(e_1)\}$. This is finally returned. The partial order consistent with it is: e_1 and then e_2.

Table 2 shows how the axioms for the Sagas transaction model [17], that were formulated in [6] using the ACTA formalism, can be expressed using the $<$ and $\rightarrow$ primitives. A Saga, S, is a sequence of sub-transactions, T_i, $i = 1, \ldots, n$. The term 'post' denotes the postcondition of the given event. The Saga commits iff all subtransactions are successfully executed in the specified order; otherwise, if one of the subtransactions aborts, the Saga aborts and the compensating transactions CT_i are executed in the reverse order. Since the specifications use only the $<$ and $\rightarrow$ primitives, our scheduler can be used to execute relaxed transactions with Sagas semantics.

Table 2. SAGA dependencies in ACTA and in the $<, \rightarrow$ notation.

	ACTA	$<, \rightarrow$ notation
post(begin(S))	T_i $\mathcal{BCD}$ T_{i-1}	$st(T_i) \rightarrow cm(T_{i-1}) \wedge cm(T_{i-1}) < st(T_i)$
	CT_j $\mathcal{WCD}$ CT_{j+1}	$cm(CT_{j+1}) < st(CT_j)$
	CT_{n-1} $\mathcal{BAD}$ S	$st(CT_{n-1}) \rightarrow ab(S) \wedge ab(S) < st(CT_{n-1})$
post(begin(T_i))	S $\mathcal{AD}$ T_i	$ab(T_i) \rightarrow ab(S)$
	T_i $\mathcal{WD}$ S	$cm(T_i) < ab(S)$
	CT_i $\mathcal{BCD}$ T_i	$st(CT_i) \rightarrow cm(T_i) \wedge cm(T_i) < st(CT_i)$
post(commit(T_i))	CT_i $\mathcal{CMD}$ S	$ab(S) \rightarrow cm(CT_i)$
	CT_i $\mathcal{BAD}$ S	$st(CT_i) \rightarrow ab(S) \wedge ab(S) < st(CT_i)$
post(begin(T_n))	S $\mathcal{SCD}$ T_n	$cm(T_n) \rightarrow cm(S)$

6. Executing multidatabase transactions

Three issues in executing multidatabase transactions are: concurrency control, safety and recoverability.

6.1. Concurrency control

Our scheduler is part of a multidatabase environment in which local database systems (LDBS) cooperate in the execution of global transactions. Each LDBS will, in general, contain a concurrency control module, which enforces local concurrency control (typically ensuring local serializability). We may assume that a task executing at each of the local systems has a *serialization event* that determines its position in the local serialization order. For example, if the local system uses two-phase locking (2PL), the serialization order of a local transaction is determined by its lock point—the point when the last lock of the transaction is granted.

A problem arises if local concurrency control modules impose an inconsistent ordering on serialization events of tasks belonging to a given multidatabase application. We resolve this problem by transferring the responsibility for global concurrency control to the scheduler. This is achieved by restating the concurrency control obligations as a set of dependencies, which are then processed like other dependencies. Unlike other scheduling dependencies, concurrency control dependencies arise at run-time, when a serialization precedence between tasks in different applications is established at some site. However, once these dependencies are added, they can be processed as usual. Thus we have a uniform mechanism for both dependency enforcement and concurrency control.

The main difficulty in this approach is that the serialization events are neither reported by the local concurrency controllers, nor can they be deduced from the temporal order of other significant events controlled by the global scheduler (start, commit, terminate). It is possible for a local concurrency controller to completely execute task T_i before task T_j has even begun, yet serialize them in such a way that that T_j precedes T_i. This problem can be overcome by using the idea of *tickets* introduced in [19]. As in [19], we may add a ticket read and ticket write operation to each task of a global application. These ticket read/write operations can be regarded as significant events, and so their execution can be controlled by declaring dependencies that refer to them. Thus the required concurrency control is obtained simply by declaring an appropriate set of ticket access dependencies.

6.2. Flexible transaction safety

A *flexible transaction* [12] is defined as a set of subtransactions and their scheduling preconditions along with a set of conditions over their final states [12]. These conditions specify the *acceptable termination states* of the flexible transaction; it completes successfully iff it terminates in such a state.

Consider the following example, adapted from [22]. We have a travel agent flexible transaction, consisting of reserve-flight (F) and reserve-car (C) subtransactions. If we fail to secure a car reservation, we wish to cancel the plane reservation. This cancellation is achieved by a subtransaction F^-, which is a *compensating transaction* for F. Thus the set of acceptable termination states for the overall transaction is given in table 3, where *in*, *cm* and *ab* indicate that the subtransaction is in its initial state, is committed and is aborted, respectively. The set of acceptable states is a constraint on the execution of a flexible transaction. This constraint can also be expressed as the set of dependencies given in table 3.

Table 3. Acceptable states of a flexible transaction.

F	F^-	C
cm	in	cm
ab	in	in
ab	in	ab
cm	cm	ab
in	in	in
cm	cm	in

$ab_C < cm_{F^-}$
$(ab_C \wedge cm_F) \rightarrow cm_{F^-}$
$cm_C \rightarrow cm_F$

6.3. Recoverability

The above approach provides a sound basis for specifying the error recovery criteria for complex applications.

Compensation dependencies are an example of semantic error recovery criteria, because they enable us to initiate additional tasks when a component of the application fails.

Another set of recovery issues arise when the scheduler itself may fail. The following data must be checkpointed in order to enable recovery of the scheduler from a failure:

(i) The current state of every dependency automaton.
(ii) Any (partially executed) pathset (see section 5), plus the current state along every path in the pathset.
(iii) The set of pending events.

The above data are subject to concurrent updates that must be executed atomically with respect to the checkpointing mechanism. For example, when an event ε is executed, the current state of every dependency automaton A_D where ε occurs in D must be updated. We do not wish a checkpoint to reflect only some of these updates. It should either reflect none of them (corresponding to a state before ε is executed), or reflect all of them (corresponding to a state after ε is executed).

In addition, the communication mechanism between the scheduler and the tasks must be persistent, so that no messages are lost while the scheduler is down (i.e., after a failure and before recovery from that failure). Mailboxes or persistent pipes may be used to provide this functionality.

7. Conclusions and future work

We addressed the problem of specifying and enforcing intertask dependencies. Our framework allows dependencies to be stated modularly and succinctly as constraints across tasks. The actual set of significant events and task state transition diagrams is not predetermined, but can vary with the application. Our framework can be extended to accommodate the issues of concurrency control, flexible transaction safety, recoverability, and the enforcement of other dependencies that are introduced dynamically at run-time (e.g., see [25]).

We showed how a dependency can be expressed as an automaton that captures all the computations that satisfy the dependency. We presented a scheduling algorithm that simultaneously enforces multiple dependencies. We showed that every global computation generated by the scheduler satisfies all of the dependencies. We also showed how relaxed transaction models such as the Saga model can be captured in our framework. The desiderata for a task scheduler for multidatabase transaction processing include correctness (no dependencies are violated), safety (transaction terminates only in an acceptable state), recoverability, and optimality and quality. We have established the correctness, safety and recoverability of the scheduler; we defer to future work issues concerning the quality of the schedules generated and the optimality of generating them.

Our approach assumes that finite state automata are available for each dependency and is independent of how those automata are produced. The automata can readily be produced manually for the common dependencies. We are developing a synthesis technique for automatically synthesizing an automaton for a dependency from the CTL formula for the dependency. This technique extends the CTL synthesis technique of [14]. Our synthesis procedure requires only the specification of the dependencies, *not* of the tasks over which those dependences are defined. That is, the precise transitions for a task's state transition diagram do not affect the representations of the different dependencies. As a result, our procedure generates an *open* system. By contrast, traditional temporal logic synthesis methods [14, 27] require a specification of the entire system. Thus their results have to be recomputed whenever the system is modified. Our synthesis procedure also takes the event attributes into account. However, this aspect has not yet been fully formalized and its details remain beyond the scope of the present paper. Rigorous synthesis techniques are a major open problem that we defer to future work.

An implementation of this work has been completed as part of the distribution services of the Carnot project at MCC. Our implementation is in the concurrent actor language Rosette [32], whose asynchrony and other features make for a natural realization of our execution model. Key features of our intertask dependency framework have also been adopted by the METEOR system.

Acknowledgments

We are indebted to Greg Meredith and Christine Tomlinson for discussions. We also benefited from conversations with Phil Cannata and Darrell Woelk. Discussions of this paper at ETH-Zürich during Amit Sheth's visit, and comments by H Ye were helpful. Sridhar Ganti provided the Sagas example.

Appendix A. CTL

We have the following syntax for CTL (where p denotes an atomic proposition, and f, g denote (sub-)formulae):

(i) Each of p, $f \wedge g$ and $\neg f$ is a formula (where the latter two constructs indicate conjunction and negation, respectively).
(ii) $EX_j f$ is a formula that intuitively means that there is an immediate successor state reachable by executing one step of process P_j in which formula f holds.
(iii) $A[fUg]$ is a formula that intuitively means that for every computation path, there is some state along the path where g holds, and f holds at every state along the path until that state.
(iv) $E[fUg]$ is a formula that intuitively means that for some computation path, there is some state along the path where g holds, and f holds at every state along the path until that state.

Appendix A.1. Formal semantics

We give the formal semantics of CTL formulae with respect to a structure $M = (S, A_1, \ldots, A_k, L)$ that consists of:
S—a countable set of states
A_i—$\subseteq S \times S$, a binary relation on S giving the possible transitions by process i, and
L—a labelling of each state with the set of atomic propositions true in the state.

Let $A = A_1 \cup \cdots \cup A_k$. We require that A be total, i.e., that $\forall x \in S, \exists y : (x, y) \in A$. A *fullpath* is an infinite sequence of states $(s_0, s_1, s_2 \ldots)$ such that $\forall i (s_i, s_{i+1}) \in A$. To any structure M and state $s_0 \in S$ of M, there corresponds a computation tree (whose nodes are labelled with occurrences of states) with root s_0 such that $s \xrightarrow{i} t$ is an arc in the tree iff $(s, t) \in A_i$.

$M, s_0 \models f$ means that f is true at state s_0 in structure M. When the structure M is understood, we write $s_0 \models f$. $\models$ is defined inductively:

$s_0 \models p$ iff $p \in L(s_0)$
$s_0 \models \neg f$ iff not$(s_0 \models f)$
$s_0 \models f \wedge g$ iff $s_0 \models f$ and $s_0 \models g$
$s_0 \models EX_j f$ iff for some state t, $(s_0, t) \in A_j$ and $t \models f$,
$s_0 \models A[fUg]$ iff for all fullpaths $(s_0, s_1, \ldots)$, $\exists i [i \geq 0 \wedge s_i \models g \wedge \forall j (0 \leq j \wedge j < i \Rightarrow s_j \models f)]$
$s_0 \models E[fUg]$ iff for some fullpath $(s_0, s_1, \ldots)$, $\exists i [i \geq 0 \wedge s_i \models g \wedge \forall j (0 \leq j \wedge j < i \Rightarrow s_j \models f)]$

We introduce the abbreviations $f \vee g$ for $\neg(\neg f \wedge \neg g)$, $f \Rightarrow g$ for $\neg f \vee g$ and $f \equiv g$ for $(f \Rightarrow g) \wedge (g \Rightarrow f)$ for logical disjunction, implication and equivalence, respectively. We also introduce a number of additional modalities as abbreviations: AFf for $A[trueUf]$, EFf for $E[trueUf]$, AGf for $\neg EF \neg f$, EGf for $\neg AF \neg f$, $AX_i f$ for $\neg EX_i \neg f$, EXf for $EX_1 f \vee \cdots \vee EX_k f$, AXf for $\neg EXf$.

Particularly useful modalities are AFf, which means that for every path, there exists a state on the path where f holds, and AGf, which means that f holds at every state along every path. Thus, AFf is an *eventuality* formula, which corresponds to a liveness property and makes a promise that something will happen. Similarly, AGf is an *invariance* formula, which corresponds to a safety property and asserts that certain conditions will hold throughout.

Appendix A.2. Expressing dependencies in CTL

Atomic propositions naturally model the states of a given system: each proposition corresponds to a significant event and holds in the state immediately following the occurrence of that event.

Now we show how certain dependencies that were motivated and defined by other researchers can be expressed uniformly in CTL.

- Order dependency [23]: if both events e_1 and e_2 occur, then e_1 precedes e_2. This was expressed as $e_1 < e_2$ in the above discussion. In CTL, it becomes:

$$AG[e_2 \Rightarrow AG \neg e_1].$$

  That is, if e_2 occurs, then e_1 cannot occur subsequently.
- Existence dependency [23]: if event e_1 occurs sometimes, then event e_2 also occurs sometimes. This was expressed as $e_1 \rightarrow e_2$ in the above discussion. In CTL, it becomes:

$$\neg E[\neg e_2 U (e_1 \wedge EG \neg e_2)].$$

  That is, there is no computation such that e_2 does not occur until a state s is reached where s satisfies $(e_1 \wedge EG \neg e_2)$, i.e., e_1 is executed in state s, and subsequently, e_2 never occurs.

The following instances of the above dependencies have also appeared in the literature.

- Commit dependency [6]: transaction A is commit-dependent on transaction B, iff if both transactions commit, then A commits before B commits. Let the relevant significant events be denoted as cm_A and cm_B:

$$AG[cm_B \Rightarrow AG \neg cm_A].$$

- Abort dependency [6]: transaction A is abort-dependent on transaction B, iff if B aborts, then A must also abort. Let the significant events here be ab_A and ab_B, so this can be written $ab_B \rightarrow ab_A$, and is rendered in CTL just like $e_1 \rightarrow e_2$ above:

$$\neg E[\neg ab_A \; U \; (ab_B \wedge EG \neg ab_A)].$$

- Conditional existence dependency [23]: if event e_1 occurs, then if event e_2 also occurs, then event e_3 must occur. That is, the existence dependency between e_2 and e_3 comes into force if e_1 occurs. This can be written $e_1 \rightarrow (e_2 \rightarrow e_3)$. Translating it to CTL involves two applications of the translation of $e_1 \rightarrow e_2$ given above, one nested inside the other. The first application, to $e_2 \rightarrow e_3$, yields the following 'mixed' formula:

$$e_1 \rightarrow \neg E[\neg e_3 \; U \; (e_2 \wedge EG \neg e_3)].$$

  The second application, which substitutes $\neg E[\neg e_3 U (e_2 \wedge EG \neg e_3)]$ for e_2 in the CTL translation of $e_1 \rightarrow e_2$ given above, gives us

$$\neg E[\; \neg\neg E[\neg e_3 U (e_2 \wedge EG \neg e_3)] \; U$$

$$(e_1 \wedge EG \neg\neg E[\neg e_3 U (e_2 \wedge EG \neg e_3)]) \;].$$

  Eliminating the double negations finally yields the following formula:

$$\neg E[\; E[\neg e_3 U (e_2 \wedge EG \neg e_3)] \; U$$

$$(e_1 \wedge EGE[\neg e_3 U (e_2 \wedge EG \neg e_3)]) \;].$$

Appendix B. Correctness of the pathset search algorithm

We now establish some correctness properties of the pathset search algorithm. We use Hoare logic [21] to express partial correctness properties, i.e., {Pre} Proc {Post} means that if procedure Proc is called with arguments that satisfy Pre, then if Proc terminates, it does so in a state that satisfies Post.

Hoare logic We use the following fragment of Hoare logic:

Backward assignment axiom:

$$\{P\langle E/X\rangle\}X := E\{P\}$$

where $\langle E/X\rangle$ denotes substitution of expression E for variable X.

Rule of consequence:

$$\frac{P \Rightarrow P',\quad \{P'\}S\{Q'\},\quad Q' \Rightarrow Q}{\{P\}S\{Q\}}$$

where S is any program statement.

Lemma 2. get_candidate_paths(A, ES_a $\cap$ Σ_A, ES_r $\cap$ Σ_A, PS, s_A) returns a set of paths P in A such that the following hold. Here p is any path in P.

(i) P has s_A as a source state
(ii) For path p′ of A, if p′ $\in$ PS, then p is an extension of p′
(iii) p is executable
(iv) p is order-consistent with PS
(v) For every $\varepsilon \in$ ES_a $\cap$ Σ_A, $a(\varepsilon)$ occurs along p
(vi) For every $\varepsilon \in$ ES_r $\cap$ Σ_A, $r(\varepsilon)$ occurs along p. □

We now define the following predicates:

- Oce: o_cons(PS) $\wedge$ exec(PS)
- Acc: a_events(PS) $\subseteq$ (ES_a $\cup$ ES_Done_a)
- Rej: r_events(PS) $\subseteq$ (ES_r $\cup$ ES_Done_r)
- Due_a: ($\forall\varepsilon \in$ ES_Done_a : depset(ε) − automata_in(PS, a(ε)) $\subseteq$ AS_due)
- Due_r: ($\forall\varepsilon \in$ ES_Done_r : depset(ε) − automata_in(PS, r(ε)) $\subseteq$ AS_due)
- Term: PS $\neq \emptyset \Rightarrow$ (ES_a $\cup$ ES_r = $\emptyset \wedge$ AS_due = $\emptyset$).

The following propositions are used to prove that search_PS returns viable pathsets (lemma 10).

Proposition 3. {Oce} search_PS {Oce}.

Proposition 4. {Acc} search_PS {Acc}.

Proposition 5. {Rej} search_PS {Rej} .

Proposition 6. {Due_a} search_PS {Due_a $\vee$ $PS = \emptyset$}.

Proposition 7. {Due_r} search_PS {Due_r $\vee$ $PS = \emptyset$} .

Proposition 8. {true} search_PS {Term}.

In proposition 9 below, X and Y are constants.

Proposition 9. {ES_a = X $\wedge$ ES_Done_a $\supseteq$ Y} search_PS { ES_Done_a $\supseteq$ (X $\cup$ Y)}.

Propositions 3 to 9 are established using Hoare logic. As an example, we give a proof of proposition 6 in appendix C.

Lemma 10. For any event, ε, and global state s, if search_PS($\emptyset$, {ε}, $\emptyset$, $\emptyset$, $\emptyset$, $\emptyset$, s) terminates with PS $\neq \emptyset$, then PS is viable (w.r.t. global state s).

Proof. We show that each of the clauses of the definition of viable (definition 13) is satisfied.

The given arguments satisfy Oce, so by proposition 3, Oce holds upon termination of search_PS. By the same argument, but using propositions 4, 5, 6, 7, respectively, we establish that Acc, Rej, (Due_a $\vee$ $PS = \emptyset$) and (Due_r $\vee$ $PS = \emptyset$) also hold upon termination of search_PS.

Since Oce holds upon termination of search_PS, PS is order-consistent and executable upon termination of search_PS (be definition of Oce).

By proposition 8 and the assumption of PS $\neq \emptyset$, we have (ES_a $\cup$ ES_r = $\emptyset \wedge$ AS_due = $\emptyset$) upon termination of search_PS. Together with Acc: a_events(PS) $\subseteq$ (ES_a $\cup$ ES_Done_a), this yields a_events(PS) $\subseteq$ ES_Done_a. Since PS $\neq \emptyset$ by assumption, Due_a holds upon termination of search_PS.

By Due_a: ($\forall\varepsilon \in$ ES_Done_a : depset(ε) − automata_in(PS, a(ε)) $\subseteq$ AS_due), and AS_due = $\emptyset$ (established above), we have ($\forall\varepsilon \in$ ES_Done_a : depset(ε) $\subseteq$ automata_in(PS, a(ε))). Together with a_events(PS) $\subseteq$ ES_Done_a (established above), this gives us ($\forall\varepsilon \in$ a_events(PS) : depset(ε) $\subseteq$ automata_in(PS, a(ε))). This implies that PS is a-closed.

The proof that PS is r-closed follows exactly the same argument as the proof of a-closure given above, but uses Rej and Due_r in place of Acc and Due_a, respectively.

The get_candidates_path function is always called with global state s as its last argument, and, by lemma 2, returns a set of candidate paths all of whom have s as their source state. Since the paths in PS are only taken out of the sets of candidate paths that get_candidates_path returns, we conclude that PS has s as its source state. Thus we have proven that PS is viable. □

Lemma 11. search_PS($\emptyset$, {ε}, $\emptyset$, $\emptyset$, $\emptyset$, $\emptyset$, s) terminates for any event ε. □

The proof of lemma 11 requires some preliminary definitions and propositions. We define:

$$\begin{aligned} \text{state}(A_D, s, PS) &= s{\uparrow}D \qquad \text{if } A_D \notin \text{automata_in}(PS) \\ &= \text{last(p)} \quad \text{where p} = \text{path_of}(A_D, \text{PS}) \\ &\qquad \text{if } A_D \in \text{automata_in}(PS) \end{aligned}$$

and

$$\text{rem}(A_D, s) = A'_D,$$

where $s' \xrightarrow{\sigma} s''$ is an edge in A'_D iff:

- $s' \xrightarrow{\sigma} s''$ is an edge in A_D
- there exists a path from s to s' in A_D

in other words, A'_D is the subautomaton of A_D whose states are all the states of A_D that are reachable from s. A'_D represents that portion of A_D still to be searched by search_PS. Finally, we define:

$$vf(s, PS) = \sum_D \mid rem(A_D, state(A_D, s, PS)) \mid$$

where $|A|$ is the number of edges in automaton A, and D ranges over all of the dependencies being enforced.

Proposition 12. $vf(s, PS)$ decreases on each recursive call of search_PS.

Proof. s remains constant, and PS has some (non-empty) path $p_{D'}$ added to it, where D' is some particular dependency. There are two cases:

- PS does not contain a path from $A_{D'}$.
- PS already contains a path $p'_{D'}$ from $A_{D'}$. In this case, the get_candidate_paths function will return a set of proper extensions of $p'_{D'}$, and $p_{D'}$ is then selected from this set.

In either case, $|rem(A_{D'}, state(A_{D'}, s, PS))|$ decreases, while $|rem(A_D, state(A_D, s, PS))|$ remains the same for all dependencies D other than D'. Thus $vf(s, PS)$ decreases. □

Proposition 13. An infinite chain of recursive calls of search_PS does not arise.

Proof. By proposition 12, $vf(s, PS)$ decreases on each recursive call. Thus the occurrence of an infinite chain of recursive calls of search_PS would cause $vf(s, PS)$ to eventually become less than zero. But this is impossible, since, by the definition of $vf(s, PS)$, we see that $vf(s, PS) \geq 0$ for all values of s and PS. □

We are now ready to provide the proof of lemma 11.

Proof of lemma 11. There are only two possibilities for non-termination of search_PS:

- An infinite chain of recursive calls of search_PS occurs.
- The (only) **while** loop does not terminate.

The first case does not arise due to proposition 13. The second case does not arise due to the following argument. Every dependency automaton is finite. Thus, the get_candidate_paths function always returns a finite set **P**. Further, one element is removed from **P** on each iteration. It follows that the termination condition of the **while** loop, namely $\mathbf{P} = \emptyset$, is eventually met, causing termination of the **while** loop. □

Appendix C. Proof of proposition 6

We repeat the statement of proposition 6:

{Due_a} search_PS {Due_a $\vee$ $PS = \emptyset$}

where Due_a $\stackrel{\text{df}}{=}$ ($\forall \varepsilon \in$ ES_Done_a : depset(ε) − automata_in(PS, a(ε)) $\subseteq$ AS_due).

The proof of this is given by the annotated version of search_PS shown in figure A1, along with the accompanying arguments for each annotation. We use a two-part name for the annotations, where the first part gives a symbolic name for the predicate (which is then given in the accompanying arguments), and the second part is an instance number, so that different annotations consisting of the same predicate can be distinguished (e.g., Due_a:1 and Due_a:2 below).

Due_a:1

Assumed as a precondition.

Due_a:2

Carried forward from Due_a:1, since no variables have been changed.

Due_a:3

Carried forward from Due_a:1, since no variables have been changed.

{Due_a:3} **AS_due := AS_due ∪ depset(ES_a ∪ ES_r);** {C1:1}

where C1 $\stackrel{\text{df}}{=}$ Due_a ∧ (depset(ES_a) $\subseteq$ AS_due).

Due_a is preserved since the assignment only changes AS_due, and it does not make AS_due smaller. (depset(ES_a) $\subseteq$ AS_due) is established as a postcondition since depset(ES_a) $\subseteq$ depset(ES_a ∪ ES_r).

C1':1 where C1' $\stackrel{\text{df}}{=}$ ($\forall \varepsilon \in$ ES_Done_a ∪ ES_a : depset(ε) − automata_in(PS, a(ε)) $\subseteq$ AS_due)

We prove $C1 \Rightarrow C1'$, which justifies this annotation.

Due_a ∧ (depset(ES_a) $\subseteq$ AS_due) /* definition of C1 */

$\Rightarrow$

Due_a ∧ ($\forall \varepsilon \in$ ES_a : depset(ε) $\subseteq$ AS_due)

$\Rightarrow$ /* depset(ε) − automata_in(PS, a(ε)) $\subseteq$ depset(ε) */

Due_a ∧ ($\forall \varepsilon \in$ ES_a : depset(ε) − automata_in(PS, a(ε)) $\subseteq$ AS_due)

$\Rightarrow$ /* by definition of Due_a and elementary set theory */

($\forall \varepsilon \in$ ES_Done_a ∪ ES_a : depset(ε) − automata_in(PS, a(ε)) $\subseteq$ AS_due)

{C1':1} **ES_Done_a := ES_Done_a ∪ ES_a;** {C2:1} where C2 $\stackrel{\text{df}}{=}$ Due_a.

Substituting ES_Done_a ∪ ES_a for ES_Done_a in C2 yields C1', thus the annotation is valid by the assignment axiom for Hoare logic.

{C2:2} **AS_due := AS_due − {A};** {C3:1}

where C3 $\stackrel{\text{df}}{=}$ ($\forall \varepsilon \in$ ES_Done_a : depset(ε) − automata_in(PS, a(ε)) $\subseteq$ AS_due ∪ {A}).

If A $\notin$ AS_due holds before the assignment, then AS_due is unchanged by the assignment, and so C3 holds afterwards since AS_due $\subseteq$ AS_due ∪ {A}. If A $\in$ AS_due holds before the assignment, then AS_due ∪ {A} after the assignment has the same value as AS_due before the assignment, thus C3 is established.

{C3:2} **if A ∈ automata_in(PS) then PS := PS − {path_of(A, PS)};** {C3:3}.

If A $\notin$ automata_in(PS) holds before the test, then no variables are changes, and so C3 is trivially preserved. Otherwise, depset(ε) − automata_in(PS, a(ε)) either remains the same or has A added as an element. But since A is an element of AS_due ∪ {A}, the relation depset(ε) − automata_in(PS, a(ε)) $\subseteq$ AS_due ∪ {A} is preserved.

{C3:4}

We can carry this annotation into the while loop at each iteration because the unprimed variables remain unchanged from one iteration to the next. The only time unprimed variables are changed is immediately before the **return** statement, which causes termination of the loop. Thus an assertion over only the unprimed variables which holds upon entry into the loop will continue to hold at the beginning of every iteration.

{C3:4} **AS_due' := AS_due;** {C4:1}

where C4 $\stackrel{\text{df}}{=}$ ($\forall \varepsilon \in$ ES_Done_a : depset(ε) −

```
search_PS(AS_due, ES_a, ES_r, PS, ES_Done_a, ES_Done_r, Curr_St)

{Duc_a:1}
if (ES_a ∪ ES_r) = ∅ and AS_due = ∅ then
  return;
    {Duc_a:2}
else {  /* Add to the automata set */
  {Duc_a:3} AS_due := AS_due ∪ depset(ES_a ∪ ES_r); {C1:1}
  {C1':1} ES_Done_a := ES_Done_a ∪ ES_a; {C2:1}
  ES_Done_r := ES_Done_r ∪ ES_r;
  A := select_a(AS_due);        /* select an automaton */
  {C2:2} AS_due := AS_due − {A}; {C3:1}
  P := get_candidate_paths(A, ES_Done_a ∩ Σ_A, ES_Done_r ∩ Σ_A, PS, Curr_St ↑ A);
  {C3:2} if A ∈ automata_in(PS) then PS := PS − {path_of(A, PS)}; {C3:3}
  while P ≠ ∅
     p := select_p(A, P);
     /* check path p */
     P := P − {p};
     {C3:4} AS_due' := AS_due; {C4:1}
     ES_a' := a_events({p}) − ES_Done_a;
     ES_r' := r_events({p}) − ES_Done_r;
     {C4:2} PS' := PS ∪ {p}; {C5:1}
     {C5:2} ES_Done_a' := ES_Done_a; {C6:1}
     ES_Done_r' := ES_Done_r;
     {C6:2} search_PS(AS_due', ES_a', ES_r', PS', ES_Done_a', ES_Done_r', Curr_St); {C7:1}
     if PS'≠ ∅ then /* PS' is viable; end all recursive calls */
           {C8:1} AS_due := AS_due'; {C9:1}
           ES_a := ES_a';
           ES_r := ES_r';
           {C9:2} PS := PS'; {C10:1}
           {C10:2} ES_Done_a := ES_Done_a'; {C11:1}
           ES_Done_r := ES_Done_r';
           return;
             {C11:2}
  endwhile
  /* all paths in P failed, so return ∅ */
  PS := ∅;
  return;
  {PS = ∅ }
  }
```

Figure A1. Annotated pathset search algorithm.

automata_in(PS,a(ε)) ⊆ AS_due' ∪ {A}).

Substituting AS_due for AS_due' in C4 yields C3, thus the annotation is valid by the assignment axiom for Hoare logic.

{C4:2} **PS' := PS ∪ {p}**; {C5:1}

where C5 $\stackrel{df}{=}$ ($\forall\varepsilon$ ∈ ES_Done_a : depset(ε) − automata_in(PS',a(ε)) ⊆ AS_due')

Substituting PS ∪ {p} for PS' in C5 yields ($\forall\varepsilon$ ∈ ES_Done_a : depset(ε) − automata_in(PS ∪ {p},a(ε)) ⊆ AS_due'). By lemma 2, p is a path in automaton A. Thus this reduces to: ($\forall\varepsilon$ ∈ ES_Done_a : depset(ε) − (automata_in(PS,a(ε)) ∪ A) ⊆ AS_due'). This is implied by C4. Thus the annotation is valid by the assignment axiom and the rule of consequence

{C5:2} **ES_Done_a' := ES_Done_a;** {C6:1}

where C6 $\stackrel{df}{=}$ ($\forall\varepsilon$ ∈ ES_Done_a' : depset(ε) − automata_in(PS', a(ε)) ⊆ AS_due').

Substituting ES_Done_a for ES_Done_a' in C6 yields C5, thus the annotation is valid by the assignment axiom for Hoare logic.

{C6:2} **search_PS(AS_due', ES_a', ES_r', PS', ES_Done_a', ES_Done_r', Curr_St);** {C7:1}

where C7 $\stackrel{df}{=}$ C6 ∨ (PS' = ∅).

This annotation is exactly the same as proposition 6, except that primed variables have replaced unprimed variables. We assume that it holds. In other words, we establish proposition 6 by assuming that it holds for all recursive calls, and then proving that it also holds for the outermost call. Thus, this is a proof by induction on the recursion tree of procedure search_PS (see, for example, [16], page 173, rule of parametrized recursion).

{C8:1} where C8 $\stackrel{df}{=}$ C6.

Follows from C7:1 and the if-test (PS ≠ ∅).

{C8:1} **AS_due := AS_due'**; {C9:1}

where C9 $\stackrel{df}{=}$ ($\forall\varepsilon$ ∈ ES_Done_a' : depset(ε) − automata_in(PS', a(ε)) ⊆ AS_due).

Substituting AS_due' for AS_due in C9 yields C8, thus the

annotation is valid by the assignment axiom for Hoare logic.

{C9:2} **PS := PS'**; {C10:1} where C10 $\stackrel{\text{df}}{=}$ ($\forall \varepsilon \in$ ES_Done_a' : depset(ε) − automata_in(PS, a(ε)) $\subseteq$ AS_due).

Substituting PS' for PS in C10 yields C9, thus the annotation is valid by the assignment axiom for Hoare logic.

{C10:2} **ES_Done_a := ES_Done_a'**; {C11:1}

where C11 $\stackrel{\text{df}}{=}$ ($\forall \varepsilon \in$ ES_Done_a : depset(ε) − automata_in(PS, a(ε)) $\subseteq$ AS_due).

Substituting ES_Done_a' for ES_Done_a in C11 yields C10, thus the annotation is valid by the assignment axiom for Hoare logic.

{C11:2}

Carried forwards from C11:1. Note that C11 = Due_a.

References

[1] Ansari M, Ness L, Rusinkiewicz M and Sheth A 1992 Using flexible transactions to support multi-system telecommunication applications *Proc. 18th VLDB Conf. (Vancouver, 1992)* (San Mateo, CA: Morgan Kaufmann) pp 65–76

[2] Attie P, Singh M, Sheth A and Rusinkiewicz M 1993 Specifying and enforcing intertask dependencies *Proc. 19th VLDB Conf. (Dublin, 1993)* (San Mateo, CA: Morgan Kaufmann) pp 134–45

[3] Attie P and Emerson E A 1989 Synthesis of concurrent systems with many similar sequential processes *Proc 16th Ann. ACM Symp. on Principles of Programming Languages (Austin, TX, 1989)* pp 191–201

[4] Breitbart Y and Silberschatz A 1988 Multidatabase Update Issues *Proc. ACM SIGMOD Int. Conf. on Management of Data (Chicago, IL, 1988)* (New York: ACM Press) pp 134–42

[5] Cannata P E 1991 The irresistible move towards interoperable database systems *Proc. 1st Int. Workshop on Interoperability in Multidatabase Systems (Kyoto, 1991)* (Los Alamitos, CA: IEEE Press)

[6] Chrysanthis P and Ramamritham K 1994 Synthesis of extended transaction models using ACTA *ACM Trans. Database Syst.* **19** (3) 450–91

[7] Clarke E, Emerson E A and Sistla P 1983 Automatic verification of finite state concurrent systems using temporal logic specifications *Proc. 10th Ann. ACM Symp. on Principles of Programming Languages (Austin, TX)* pp 117–26 (1986 *ACM Trans. Program. Lang. Syst.* **8** 244–63)

[8] Clarke E and Grumberg O 1987 Avoiding the state explosion problem in temporal logic model checking algorithms *Proc. 6th Ann. ACM Symp. on Principles of Distributed Computing (Vancouver, 1987)* pp 294–303

[9] Clarke E and Grumberg O 1987 Research on automatic verification of finite state concurrent systems *Ann. Rev. Comput. Sci.* **2** 269–90

[10] Dayal U, Hsu M and Ladin R 1991 A transactional model for long-running activities *Proc.17th VLDB Conf. (Barcelona, 1991)* (San Mateo, CA: Morgan Kaufmann) pp 113–22

[11] Elmagarmid A (ed) 1992 *Database Transaction Models* (San Mateo, CA: Morgan Kaufmann)

[12] Elmagarmid A, Leu Y, Litwin W and Rusinkiewicz M 1990 A multidatabase transaction model for Interbase *Proc. 16th VLDB Conf. (Brisbane, 1990)* (San Mateo, CA: Morgan Kaufmann) pp 507–18

[13] Emerson E A 1990 Temporal and modal logic *Handbook of Theoretical Computer Science* vol B, ed J Van Leeuwen

[14] Emerson E A and Clarke E 1982 Using branching time temporal logic to synthesize synchronization skeletons *Science of Computer Programming* vol 2, pp 241–66

[15] Elmasri R and Navathe S 1994 *Fundamental of Database Systems* 2nd edn (Reading, MA: Benjamin Cummings)

[16] Francez N 1992 *Program Verification* (Reading, MA: Addison-Wesley)

[17] Garcia-Molina H and Salem K 1987 Sagas *Proc. ACM SIGMOD Conf. on Management of Data*

[18] Georgakopoulos D, Hornick M and Sheth A 1995 An overview of workflow management: from process modeling to workflow automation infrastructure *Distrib. Parallel Databases* **3** (2) 119–53

[19] Georgakopoulos D, Rusinkiewicz M and Sheth A 1991 On serializability of multidatabase transactions through forced local conflict *Proc. 7th Int. Conf. on Data Engineering (Kobe, 1991)* (Los Alamitos, CA: IEEE Computer Society Press) pp 314–23

[20] Gray J and Reuter A 1993 *Transaction Processing: Concepts and Techniques* (San Mateo, CA: Morgan Kaufmann)

[21] Hoare C A R 1969 An axiomatic basis for computer programming *Commun. ACM* **12** 576–80, 583

[22] Jin W, Ness L, Rusinkiewicz M and Sheth A 1992 Executing service provisioning applications as multidatabase flexible transactions *Bellcore Technical Memorandum*

[23] Klein J 1991 Advanced rule driven transaction management *Proc. IEEE COMPCON (1991)*

[24] Korth H F and Speegle G 1994 Formal aspects of concurrency control in long-duration transaction systems using the NT/PV model *ACM Trans. Database Syst.* **19** 492–535

[25] Krishnakumar N and Sheth A 1995 Managing heterogeneous multi-system tasks to support enterprise-wide operations *J. Distrib. Parallel Database Syst.* **3** 155–86

[26] Leymann F, Schek H-J and Vossen G (ed) 1996 Transactional workflows *Dagstuhl Seminar Report 152* IBFI GmbH, Schloss Dagstuhl, Wadern

[27] Manna Z and Wolper P 1984 Synthesis of communicating processes from temporal logic specifications *ACM Trans Program. Lang. Syst.* **6** 68–93

[28] Ramamritham K and Chrysanthis P 1996 A taxonomy of correctness criteria in database applications *VLDB J.*

[29] Rusinkiewicz M and Sheth A 1994 Specification and execution of transactional workflows *Modern Database Systems: The Object Model, Interoperability, and Beyond* ed W Kim (New York: ACM Press–Addison-Wesley) (Reprinted with corrections 1995)

[30] Rusinkiewicz M, Sheth A and Karabatis G 1991 Specifying interdatabase dependencies in a multidatabase environment *IEEE Comput.* **24** (12) 46–53

[31] Sheth A, Georgakopoulos D, Joosten S, Rusinkiewicz M, Scacchi W, Wileden J and Wolf A 1996 Report from the NSF Workshop on Workflow and Process Automation in Information Systems *Technical Report UGA-CS-TR-96-003* Department of Computer Science, University of Georgia. http://lsdis.cs.uga.edu/activities/NSF-workflow/

[32] Tomlinson C, Cannata P E, Meredith G and Woelk D 1993 The extensible services switch in Carnot *IEEE Parallel Distrib. Technol.* **1** 16–20

[33] Woelk D, Cannata P, Huhns M, Jacobs N, Ksiezyk T, Lavender G, Meredith G, Ong K, Shen W, Singh M and Tomlinson C 1996 Carnot prototype *Object-Oriented Multidatabase Systems* ed O Bukhres and A K Elmagarmid (Englewood Cliffs, NJ: Prentice Hall) pp 621–48

[34] Worah D and Sheth A 1996 What do advanced transaction models have to offer for workflows? *Proc. Int. Workshop on Advanced Transaction Models and Architectures (ATMA) (Goa, 1996)*