Specifying and Enforcing Intertask Dependencies

Paul C. Attie  Munindar P. Singh
Carnot Project, MCC
3500 W. Balcones Center Drive
Austin, TX 78759
USA
{attie, mSingh}@mcc.com

Amit Sheth
Bellcore
444 Hoes Lane
Piscataway, NJ 08854
USA
amit@ctt.bellcore.com

Marek Rusinkiewicz
Dept of Computer Science
University of Houston
Houston, TX 77204
USA
marek@cs.uh.edu

Abstract

Extensions of the traditional atomic transaction model are needed to support the development of multi-system applications or workflows that access heterogeneous databases and legacy application systems. Most extended transaction models use conditions involving events or dependencies between transactions. Intertask dependencies can serve as a uniform framework for defining extended transaction models. In this paper, we introduce event attributes needed to determine whether a dependency is enforceable and to properly schedule events in extended transaction models. Using these attributes and a formalization of a dependency into the temporal logic CTL, we can automatically synthesize an automaton that captures the computations that satisfy the given dependency. We show how a set of such automata can be combined into a scheduler that produces global computations satisfying all relevant dependencies. We show how dependencies required to implement relaxed transactions such as Sagas can be enforced and discuss briefly the issues of concurrency control, safety, and recoverability.

1 Introduction

One of the main objectives of the Carnot project at M-C is to provide an environment for the development of applications that access related information stored in multiple existing systems [Ca91]. An important component of this effort is a facility for relaxed task man-

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 19th VLDB, Dublin, Ireland, 1993

agement. A task is any unit of computation that performs some useful function in a system. The tasks that are of particular interest are database transactions. To efficiently develop such multi-system applications accessing existing heterogeneous and closed\(^1\) systems, we must be able to modularly capture the execution constraints of various applications. This can be achieved by modeling them as relaxed transactions consisting of related tasks executed on different systems.

The requirements of the traditional transaction model based on full isolation, atomic commitment and global serializability may be too strong, or not sufficient for a particular multi-system application. 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 [Gra81, HR83] for multi-system transactions to capture precisely the synchrony and coupling requirements based on the true application semantics. The semantic constraints may be specified as intertask dependencies, which are constraints over significant task events, such as commit and abort.

The concomitant reduction in semantic 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 a higher availability of data, better response times, and a higher throughput. The modeling of complex telecommunication applications is discussed in [ANRS92], where it is argued that many multi-system applications can be efficiently modeled and executed as relaxed transactions.

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

\(^1\) In many such systems, the data can be accessed only through the existing interfaces, even if it is internally stored under the control of a general purpose DBMS. Such systems are frequently referred to as legacy systems and the applications that access several of them are called workflows.
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 sets off an alarm if the summary falls below a preset 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 atomic multidatabase transactions that are globally serializable [BS88]. This, however, may be inefficient or even impossible, if the database interfaces do not provide visible two-phase commit facilities. Instead, we may assume that the interdatabase integrity is maintained by executing separate tasks that obey the appropriate intertask dependencies. These dependencies state that if a delete task on the bookings database commits, then a decrement-summary task should also commit. Furthermore, 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.

Figure 1: Task Graph for the Delete Booking Example

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 re-attempted 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.

The rest of the paper is organized as follows. Section 2 provides the technical and methodological background for our work and gives an example of its application. Section 3 describes how we formally specify dependencies, 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 [GS87] can be described (and hence enforced) as a set of dependencies. Section 6 briefly discusses the concurrency control, safety and recovery issues in the context of flexible transactions [JNHS91]. Some conclusions are presented in Section 7.

2 Background

The specification and enforcement of intertask dependencies has recently received much attention [CR90, DIL90, EJ92, ELL90, KI91]. Following [K191] and [CR92], we specify intertask dependencies as constraints on the occurrence and temporal order of certain significant events. Klein has proposed the following two primitives [K191]:

1. $e_1 \rightarrow e_2$: If $e_1$ occurs, then $e_2$ must also occur. There is no implied ordering on the occurrences of $e_1$ and $e_2$.

2. $e_1 < e_2$: If $e_1$ and $e_2$ both occur, then $e_1$ must precede $e_2$.

Well-known examples of dependencies include:

- **Commit Dependency** [CR92]: 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** [CR92]: Transaction $A$ is abort-dependent on transaction $B$, iff if $B$ aborts, then $A$ aborts.

---

2 This paper is a revised and abbreviated version of the report [ASR92] available from the authors. The report contains proofs of all theorems.
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 [KL91]: 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 occurrences 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 above primitives can capture many of the semantic constraints encountered in practice: any useful framework for intertask dependencies should be at least as powerful. Our approach meets this criterion: $\rightarrow$ and $<$ are special cases of our formalism.

![Figure 2: An Example Task State Transition Diagram](image)

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. The execution of an event causes a transition of the task to another state. Figure 2 shows an example task state transition diagram taken from [KL91]. 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"). When a task is done, it can either commit, i.e., make the $cm$ transition, or abort, i.e., make the $ab$ transition.

Using the state transition diagrams and significant events defined above, we can represent the travel agent application 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).

![Figure 3: Dependencies Between Significant Events in the Delete Booking Example](image)

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 re-executed, a new id may be generated for it (and for all its events). The dependencies can be appropriately modified and everything can proceed normally.

Let $e, e_i, e_j, \text{etc.}$ denote any significant event and $D(e_1, \ldots, e_n)$ denote an unspecified dependency over $e_1, \ldots, e_n$.

3.1 Formal Specification of Dependencies

We adopt the language of Computation Tree Logic (CTL) as the language of our dependencies [EM90]. CTL is a powerful language, well-known from distributed computing. A brief description of CTL and modeling 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. And, 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.

We would like our dependencies to be easily specifi-
able 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.

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 below shows the attributes of the significant events of transactions commonly found in database applications and DBMSs. Therein, an √ indicates that the given attribute always holds, whereas a × indicates that the given attribute may not always hold.

<table>
<thead>
<tr>
<th>Event</th>
<th>Forcible?</th>
<th>Rejectable?</th>
<th>Delayable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>cm</td>
<td>×</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>ab</td>
<td>√</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>pr</td>
<td>×</td>
<td>√</td>
<td>×</td>
</tr>
<tr>
<td>st</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
</tbody>
</table>

Table 1: Attribute Tables for Significant Events

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 create a finite state machine $A_D$ that is responsible for enforcing $D$. $A_D$ captures all possible orders of event on which $D$ is satisfied. This can be done either manually, or by using an extension of the CTL synthesis technique of [EC82, Em90] that we have developed [ASRS92]. Our procedure requires only the specification of the dependencies, not of the tasks over which those dependencies 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 [EC82, MW84] require a specification of the entire system. Thus their results have to be recomputed whenever the system is modified. The details of the synthesis procedure are omitted for brevity, but can be found in [ASRS92]. In the worst case, the size of $A_D$ is exponential in the number of events.
in $D$. This number is often small (in our experience, 2–4), so the complexity is not a major impediment in practice.

$A_D$ is a tuple $(s_0, S, \Sigma, \rho)$, where $S$ is a set of states, $s_0$ is the distinguished initial state, $\Sigma$ 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 $e$ to denote any event which can either be a significant event (notated with $e$) or a termination event. We discuss the generation and usage of termination events below. The elements of $\Sigma$ are notated as $\sigma$, $\sigma'$, etc. $\sigma$ can be of any of the forms described below.

- $a(\varepsilon_1, \ldots, \varepsilon_m)$: This indicates that $A_D$ accepts the events $\varepsilon_1$ through $\varepsilon_m$. If this transition is taken by $A_D$, then each $\varepsilon_i$ is accepted and, if $\varepsilon_i$ is a significant event, it is then forwarded to the event monitor for execution.

- $r(\varepsilon_1, \ldots, \varepsilon_m)$: This indicates that $A_D$ rejects the events $\varepsilon_1$ through $\varepsilon_m$ 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 $\sigma_i$ through $\sigma_n$.

- $\sigma_1;\ldots;\sigma_n$, where the $\sigma_i \in \Sigma$: This indicates the accept operations of $\sigma_i$ occur before the accept operations of $\sigma_{i+1}$ (for $1 \leq i \leq (n - 1)$).

Example Dependency Automata

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

In Figures 4 and 5 below, we give example dependency automata for the dependencies $\varepsilon_1 < \varepsilon_2$, and $\varepsilon_1 \rightarrow \varepsilon_2$, respectively. The symbol $|$ indicates choice: an edge labeled $\sigma|\sigma'$ may be followed if the scheduler permits either $\sigma$ or $\sigma'$.

The Operation of an Automaton

We assume for simplicity that each task can have at most one event in a given dependency, i.e., only intertask dependencies are explicitly considered. Thus the input alphabet for $A_D$, where $D$ is of the form $D(\varepsilon_1, \ldots, \varepsilon_n)$, is $\{\varepsilon_1, \ldots, \varepsilon_n, t_1, \ldots, t_n\}$. That is, the size of the input alphabet for $A_D$ is $2n$.

$A_D$ operates as follows. At any time, it is in some state, say $s$. Initially, $s = s_0$. Events arrive sequentially. Let $\varepsilon$ be the current event. If $s$ has an outgoing edge labeled $a(\varepsilon)$ 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 synchronisation with respect to the events executed so far. The behavior 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 input alphabet. So if $s$ 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. $A_t$ 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
events that may have been submitted by the given task will definitely not be submitted. This can significantly affect the automaton’s behavior. 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. This is easy enough to implement.

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 \((A_P)'s\) 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” [CG87], 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 [AE89]. 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 [ANRS92], the time complexity is polynomial. Also, the space complexity of our technique is polynomial versus the exponential 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 \( r(\epsilon) \) transition by some automata, then \( \epsilon \) is rejected, and a reject\( (\epsilon) \) message is sent to the task that submitted \( \epsilon \) 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 \( \epsilon \) 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

1. accept \( \epsilon \);
2. begin in the current global state of the scheduler;
3. be order-consistent;
4. be a-closed and r-closed; and
5. be executable.

A pathset accepts \( \epsilon \) iff all its member paths mentioning \( \epsilon \) should accept it and there should be no paths accepting the termination event associated with \( \epsilon \). 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 2–5 is called viable. 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 \( (s_D, \ldots, s_D, \ldots, s_D, \ldots) \) where \( s_D \) is the local state of \( A_D \), and \( D_1, \ldots, D_n \) are all the dependencies in the system.

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

Definition 2 (Path).
A path \( \pi_D \) 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 \( \rho_D \) 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 \( D \). Furthermore, \( A_D \) 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 is a set, \( \Pi \), of paths such that:
1. Each element of \( \Pi \) is a path in some \( A_D \).
2. Each \( A_D \) contributes at most one path to \( \Pi \).

As mentioned in Section 5.1, when an event \( \varepsilon \) is submitted to the scheduler, the scheduler attempts to execute \( \varepsilon \) by finding a viable pathset \( \Pi \) that accepts \( \varepsilon \). If such a pathset is found, then all 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 \( \varepsilon \) is placed in the pending set. Another attempt at finding a suitable pathset is made when other events affecting the acceptability of \( \varepsilon \) are submitted. Event \( \varepsilon \) remains in the pending set until a viable pathset is executed that either accepts or rejects it. In any case, the task that submitted \( \varepsilon \) is informed of this decision.

5.3 The Pathset Search Algorithm
In Figure 7, we present a (recursive) procedure \( \text{search}\_II \) that searches for viable pathsets. The procedure is initially called as \( \text{search}\_II(\emptyset) \). The event to be executed, \( \varepsilon \), and other necessary data structures are assumed to be globals for simplicity (they are passed as parameters in the actual implementation). The search procedure attempts to construct a viable pathset by selecting paths (from each relevant automaton) that are order-consistent with \( \Pi \) and are executable. If these paths contain \( a(\varepsilon) \) or \( r(\varepsilon) \) events that occur in automata outside the set of automata being considered, those automata are also considered to ensure \( a\text{-closure} \) and \( r\text{-closure} \) of the eventual solution.

The function \( \text{get\_candidate\_paths}(A, \Pi) \) returns a set of executable paths from automaton \( A \) that are order-consistent with all paths in \( \Pi \). Some of the returned paths may be extensions of paths already in \( \Pi \). We now establish some correctness properties of the pathset search algorithm. Most proofs are not included here for brevity, but appear in [ASRS92].

Lemma 1 For any event, \( \varepsilon \), and global state \( s \), if \( \text{search}\_II(\emptyset) \) terminates with \( \Pi \neq \emptyset \), then \( \Pi \) is viable (w.r.t. global state \( s \)) and accepts \( \varepsilon \).

Proof sketch.
We show that each of the clauses of the definition of viable pathsets is satisfied. The search for a pathset always begins in the current global state. New paths that are added to the candidate pathset \( (\Pi, \varepsilon) \) are ex-
search-II(II)
if r-closed(II) and a-closed(II) and II accepts ε then
  return(II);
else
  Let A be an automaton needed to close off II;
  IIc := get_candidate_paths(A, II);
  for each π ∈ IIc:
    II′ := search-II(II ∪ { π});
    if II′ ≠ φ then
      /* II′ is viable; end all recursive calls */
      return(II′);
  endfor
  /* all paths in IIc failed, so return φ */
  return(φ);

Figure 7: Pathset Search Algorithm

executable and order-consistent with II, by definition of the get_candidate_paths function. The search terminates when either II is empty or is a-closed and r-closed.

Lemma 2 search-II(φ) always terminates.

Proof sketch.
The essential idea is that because the number of automata is finite and each automaton has finitely many paths, only finitely many candidate pathsets need to be considered. Thus the algorithm terminates.

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-II(φ). If this invocation returns a nonempty II, then II is immediately executed. Otherwise, ε is placed in the pending set. II is executed by (a) accepting the events that II accepts in a partial order that is consistent with II and (b) rejecting all events rejected by II.

Definition 4 (Path Projection).
The projection η[D] of global computation η onto a dependency automaton D is the path obtained from η by removing all transitions ε such that ε ∉ ΣD.

Lemma 3 Let η be a global computation generated by the scheduler. Then, for every dependency D, η[D] is a path in AP.

Proof sketch. By construction of the scheduler.

The paths in IIc returned by get_candidate_paths are examined in arbitrary order. The quality of the generated pathset could be improved if the paths in IIc were examined according to some appropriate criterion, such as minimal length or maximal acceptance. We are currently experimenting with such criteria.

5.5 Example of Scheduler Operation

We now give an example of how relaxed transactions expressed with < and → can be scheduled using our algorithm. For simplicity, let the only dependencies in force be e1 < e2 and e1 → e2, where both e1 and e2 are rejectable and delayable. Let A_< and A_ be the corresponding automata as shown in Figures 4 and 5. Assume that e1 is submitted first. We find a(a(e1)) in A_<. However, since no path in A_ begins with a(e1), the empty pathset is returned and e1 added to the pending set. When e2 is submitted, two executable paths can be found in A_: a(e2); a(e1) and a(e1) a(e2). The a-closure requirement now forces the scheduler to search A_< for a path that accepts e1 and e2. The only such path is a(e1); a(e2). Since a(e1); a(e2) and a(e2); a(e1) are not mutually order-consistent, the only viable pathset is {a(e1); a(e2), a(e2), a(e1)}. This is finally returned. The partial order consistent with it is: e1 and then e2.

Table 2 shows how the axioms for the Saga transaction model [GS87], that were formulated in [CR92] using the ACTA formalism, can be expressed using the < and → primitives. A Saga, S, is a sequence of subtransactions, Ti, i = 1, . . . , 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 CTi are executed in the reverse order. Since the specifications use only the < and → primitives, our scheduler can be used to execute relaxed transactions with Sagas semantics.

6 Executing Multidatabase Transactions

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

6.1 Concurrency Control

Our scheduler is a 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 treated like other dependencies. However, 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, there is no difference in how they are treated. 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 [GRS91]. As in [GRS91], 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 then obtained simply by declaring an appropriate set of ticket access dependencies.

### Table 2: Expressing SAGA Dependencies in ACTA and the \(<, \rightarrow \) Notation

<table>
<thead>
<tr>
<th>( \text{post}(\text{begin}(S)) )</th>
<th>ACTA</th>
<th>(&lt;, \rightarrow ) notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_i \ BCD \ T_{i-1} )</td>
<td>( \text{st}(T_i) \rightarrow \text{cm}(T_{i-1}) \land \text{cm}(T_{i-1}) &lt; \text{st}(T_i) )</td>
<td></td>
</tr>
<tr>
<td>( CT_j \ WCD \ CT_{j+1} )</td>
<td>( \text{cm}(CT_{j+1}) &lt; \text{st}(CT_j) )</td>
<td></td>
</tr>
<tr>
<td>( CT_{n-1} \ BAC \ S )</td>
<td>( \text{st}(CT_{n-1}) \rightarrow \text{ab}(S) \land (\text{ab}(S) &lt; \text{st}(CT_{n-1})) )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \text{post}(\text{begin}(T_i)) )</th>
<th>SAD ( T_i )</th>
<th>( \text{ab}(T_i) \rightarrow \text{ab}(S) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_i \ WAD \ S )</td>
<td>( \text{cm}(T_i) &lt; \text{ab}(S) )</td>
<td></td>
</tr>
<tr>
<td>( CT_i \ BCD \ T_i )</td>
<td>( \text{st}(CT_i) \rightarrow \text{cm}(T_i) \land \text{cm}(T_i) &lt; \text{st}(CT_i) )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \text{post}(\text{commit}(T_i)) )</th>
<th>( CT_i \ CMAD \ S )</th>
<th>( \text{ab}(S) \rightarrow \text{cm}(CT_i) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( CT_i \ BCD \ S )</td>
<td>( \text{st}(CT_i) \rightarrow \text{ab}(S) \land \text{ab}(S) &lt; \text{st}(CT_i) )</td>
<td></td>
</tr>
</tbody>
</table>

| \( \text{post}(\text{begin}(T_n)) \) | SAD \( T_n \) | \( \text{cm}(T_n) \rightarrow \text{cm}(S) \) |

### Table 3: Acceptable States of a Flexible Transaction

<table>
<thead>
<tr>
<th>( F )</th>
<th>( F^- )</th>
<th>( C )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{cm} )</td>
<td>( \text{in} )</td>
<td>( \text{cm} )</td>
</tr>
<tr>
<td>( \text{ab} )</td>
<td>( \text{in} )</td>
<td>( \text{in} )</td>
</tr>
<tr>
<td>( \text{ab} )</td>
<td>( \text{in} )</td>
<td>( \text{ab} )</td>
</tr>
<tr>
<td>( \text{cm} )</td>
<td>( \text{cm} )</td>
<td>( \text{ab} )</td>
</tr>
<tr>
<td>( \text{in} )</td>
<td>( \text{in} )</td>
<td>( \text{in} )</td>
</tr>
<tr>
<td>( \text{cm} )</td>
<td>( \text{cm} )</td>
<td>( \text{in} )</td>
</tr>
</tbody>
</table>

The above data is subject to concurrent updates that must be executed atomically with respect to the checkpointing mechanism. For example, when an event \( \varepsilon \) is executed, the current state of every dependency automaton \( A_D \) where \( \varepsilon \) 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 \( \varepsilon \) is executed), or reflect all of them (corresponding to a state after \( \varepsilon \) 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).

### 6.3 Recoverability

We will not deal extensively with the issue of recovery from failure in this paper. Suffice it to say that the following data must be checkpointed in order to enable recovery of the scheduler from a failure:

1. The current state of every dependency automaton.
2. Any (partially executed) pathset (see Section 5), plus the current state along every path in the pathset.
3. The set of pending events.

The above data is subject to concurrent updates that must be executed atomically with respect to the checkpointing mechanism. For example, when an event \( \varepsilon \) is executed, the current state of every dependency automaton \( A_D \) where \( \varepsilon \) 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 \( \varepsilon \) is executed), or reflect all of them (corresponding to a state after \( \varepsilon \) 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 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.

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 enforces multiple dependencies at the same time. This algorithm uses the automata corresponding to each dependency. 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 are currently studying issues concerning the quality of the schedules generated and the optimality of generating them.

An implementation of this work has been completed as part of the distribution services of the Carnot project [Ca91] at MCC. Our implementation is in the concurrent actor language Rosette, whose asynchrony and other features make for a natural realization of our execution model. Carnot enables the development of open applications that use information stored under the control of existing closed systems. The specification and run-time enforcement of data and intertask dependencies is an important component of this effort.

Acknowledgements We are indebted to Greg Meredith and Christine Tomlinson for discussions, and to Allen Emerson for advice on CTL. We also benefited from conversations with Phil Cannata and Darrell Woelk. Discussions of this paper at ETH-Zürich and comments by H. Ye were helpful. Sridhar Ganti provided the Sagas example.

References


A  CTL Syntax and Semantics

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

1. Each of p, f \land g and \neg f is a formula (where the latter two constructs indicate conjunction and negation, respectively).
2. EXjf 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.
3. 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.
4. 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.

Formally, we give the 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 labeling 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 full path 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 labeled with occurrences of states) with root s_0 such that s_\downarrow t is an arc in the tree iff (s, t) \in A_i.

We use the usual notation to indicate truth in a structure: 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. We define \models inductively:

\begin{align*}
\models s_0 \models p & \quad \text{iff } p \in L(s_0) \\
\models s_0 \models \neg f & \quad \text{iff } s_0 \not\models f \\
\models s_0 \models f \land g & \quad \text{iff } s_0 \models f \text{ and } s_0 \models g \\
\models s_0 \models \text{EX}_j f & \quad \text{iff for some state } t, \\
& \quad (s_0, t) \in A_j \text{ and } t \models f \\
\models s_0 \models A[fUg] & \quad \text{iff for all fullpaths } (s_0, s_1, \ldots), \\
& \quad \exists[i \geq 0 \land s_i \models g \land \forall j(0 \leq j < i \Rightarrow s_j \models f)] \\
\models s_0 \models E[fUg] & \quad \text{iff for some fullpath } (s_0, s_1, \ldots), \\
& \quad \exists[i \geq 0 \land s_i \models g \land \forall j(0 \leq j < i \Rightarrow s_j \models f)]
\end{align*}

We write \models f to indicate that f is \textit{valid}, i.e., true at all states in all structures.

We introduce the abbreviations f \lor g for \neg (\neg f \land \neg g), f \Rightarrow g for f \land (f \lor g), and f \equiv g for (f \Rightarrow g) \land (g \Rightarrow f) for logical disjunction, implication, and equivalence, respectively. We also introduce a number of additional modalities as abbreviations: A[fUg] for \neg E[\neg f \lor \neg g], E[fUg] for \neg A[\neg f \lor \neg g], AFf for A[trueUf], Ef for E[trueUf], AGf for \neg Ef, EGf for \neg AFf, A[fUg] for \neg E[\neg f \lor \neg g], E[fUg] for E[\neg f \lor \neg g], AXf for \neg EXf, EXf for EXf, AXf for AXf, AFf, EGf... 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.

A formula of the form \( A[fUg] \) or \( E[fUg] \) is an eventuality formula. An eventuality corresponds to a liveness property in that it makes a promise that something does happen. This promise must be fulfilled. The eventuality \( A[fUg] \) (\( E[fUg] \)) is fulfilled for \( s \) in \( M \) provided that for every (respectively, for some) path starting at \( s \), there exists a finite prefix of the path in \( M \) whose last state satisfies \( g \) and all of whose other states satisfy \( f \). Since \( AFg \) and \( EFg \) are special cases of \( A[fUg] \) and \( E[fUg] \), respectively, they are also eventualities. In contrast, \( A[FWg] \), \( E[JWg] \) (and their special cases \( AGg \) and \( EGg \)) are invariance formulae. An invariance corresponds to a safety property since it asserts that certain conditions will necessarily be met.

CTL is a propositional branching-time temporal logic. That is, it includes propositional logic and temporal operators. A CTL temporal operator is composed of a path-quantifier (either \( A \), meaning for all possible computations, or \( E \), meaning for some possible computation), followed by a linear temporal operator (one of \( X \), \( F \), \( G \), or \( U \)). \( Xp \) means that \( p \) holds at the next point along the given computation; \( Fp \) means that \( p \) holds at some point along the given computation; \( Gp \) means that \( p \) holds at all points along the given computation; and \( pUq \) means that \( q \) holds at some point along the given computation and \( p \) holds from the current point until that point.

### A.1 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** [KI91]: 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** [KI91]: 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:
  \[
  \neg E[\neg e_2 U(e_1 \land EG\neg e_2)]
  \]
  That is, there is no computation such that \( e_2 \) does not occur until a state \( s \) is reached where \( s \) satisfies \( (e_1 \land EG\neg e_2) \), 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** [CR92]: 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** [CR92]: 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:
  \[
  \neg E[ab_A \land (ab_B \land EG\neg ab_A)]
  \]

- **Conditional Existence Dependency** [KI91]: 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:
  \[
  e_1 \rightarrow \neg E[\neg e_2 \land (e_2 \land EG\neg e_3)]
  \]
  The second application, which substitutes \( \neg E[\neg e_3 U(e_2 \land EG\neg e_3)] \) for \( e_2 \) in the CTL translation of \( e_1 \rightarrow e_2 \) given above, gives us
  \[
  \neg E[\neg E[\neg e_3 U(e_2 \land EG\neg e_3)] \land (e_1 \land EG\neg e_3)]
  \]
  Eliminating the double negations finally yields the following formula:
  \[
  \neg E[\neg e_3 U(e_2 \land EG\neg e_3)] \land (e_1 \land EGE[\neg e_3 U(e_2 \land EG\neg e_3)])
  \]

### A.2 Expressing Real-time Dependencies in CTL

We use the variant of CTL called RTCTL\(^2\) (Real-Time Computation Tree Logic ≥) [EMSS93]. This is the same as CTL except that \( EF\neg t \) means “will occur after \( t \) or more time units along some computation.”

- **Real-time Order Dependency**: If both events \( e_1 \) and \( e_2 \) occur, then \( e_1 \) precedes \( e_2 \), and \( e_2 \) occurs within \( t \) time units of \( e_1 \).
  \[
  AG[(e_2 \Rightarrow AG\neg e_1) \land (e_1 \Rightarrow \neg EF^2 t e_2)]
  \]

- **Real-time Existence Dependency**: If event \( e_1 \) occurs sometimes, then event \( e_2 \) also occurs sometimes. Furthermore, \( e_2 \) occurs no later than \( t \) time units after \( e_1 \).
  \[
  \neg E[\neg e_2 U(e_1 \land EG \neg e_2)] \land \neg EF[e_1 \land EF^2 t e_2]
  \]