Journal of Systems Integration, 5, 219–252 (1995)

A Parallel Logic Language for Transaction Specification in Multidatabase Systems

EVA KÜHN* eva@mips.complang.tuwien.ac.at
University of Technology Vienna, Institute of Computer Languages, Argentinierstraße 8, 1040 Vienna, Austria, Europe

AHMED K. ELMAGARMID ake@cs.purdue.edu
YUNGHO LEU** yhl@cs.ntit.edu.tw
Indiana Center for Database Systems, Purdue University, Department of Computer Sciences, West Lafayette, IN 47907, USA

NOUREDDINE BOUDRIGA nab@espttn.esptt.tn
CS Dept, School of Telecommunications, University of Tunis II, 2083 Ariana, Tunisia

Received November 26, 1990; Revised March 28, 1995

Editor:

Abstract. The realization of truly heterogeneous database systems is hampered by two principal obstacles. One is the unsuitability of traditional transaction models; this has led to the proposal of a number of new, advanced transaction models. The second is the lack of appropriate programming support for these advanced concepts. This paper addresses these two issues by pointing out the advantages of using a logic-based approach for the integration of autonomous software systems.

Keywords: multidatabase systems, heterogeneous transaction processing, parallel and distributed computing, Prolog, coordination languages.

1. Introduction

While significant advances have been made in the physical connection of heterogeneous and isolated data repositories into networks, the development of appropriate software to permit uniform access to these resources has lagged behind. In writing global applications, programmers must be free to concentrate on specifications directly related to the task at hand, unencumbered by larger system-integration issues such as network interfacing (connection to and reliable communication with other software systems), scheduling and load balancing, or optimization of execution plans. Multidatabase systems (MDBSs) address this problem by providing uniform access to the distributed data. An MDBS aims to make heterogeneous systems interoperable through varying levels of integration. In its most general form, the MDBS can be seen as a tool that allows the specification

* The work is supported by the Austrian FWF (Fonds zur Förderung der wissenschaftlichen Forschung), project "Multidatabase Transaction Processing", contract number P09020-MAT.
** Current address of the author: Department of Information Management, National Taiwan Institute of Technology, Taipei, Taiwan, ROC.

and synchronization of arbitrary data and control flow patterns between the distributed systems (*work flow management*) in a reliable way.

Integration consists of two aspects. First, semantic data integration compensates for the design autonomy of the local systems. Each local software system is autonomous in its underlying data model (e.g., relational, object-oriented, deductive, or hierarchical) and in the naming, structuring, and scaling of local data. Investigations into semantic data integration, particularly using an object-oriented approach, form a major focus in MDBS research. Second, global transaction management deals with the semantics and execution of MDBS transactions. As will be discussed below, traditional transaction models are insufficient, as the requirements of global transaction processing differ greatly from those of local transaction processing. For example, the treatment of communication and system failures falls under the control of the global transaction manager, which masks the failure of single components to avoid the abortion of long-lived and costly global transactions. In this paper we shall focus on this second aspect of integration.

Global and local transactions differ in that the former must maintain local system autonomy. Thus, traditional concepts applicable to local systems cannot simply be transferred to the MDBS layer. For example, local systems must access subtransactions of MDBS transactions while simultaneously serving local users, thus creating hidden indirect, conflicts between global subtransactions. To maintain global serializability in the presence of such indirect conflicts, the serialization order of *all* subtransactions of global transactions must be checked at *all* sites. This process is compromised if the commit order in the schedules of local systems does not reflect the serialization order [3]. Such anomalies complicate MDBS transaction processing and either lead to the placing of strong assumptions on the local systems to be integrated (such as the requirement of rigorous local schedulers with a visible two phase commit protocol) or require new transaction models and techniques. To overcome this anomaly, an optimistic ticket method has been proposed in [15] that enforces artificial conflicts; it has the potential drawback, however, of creating a performance bottleneck. Weaker criterions than serializability, such as quasi-serializability [9], have also been proposed for MDBS transactions.

The MDBS transaction manager must also fulfill new issues which do not arise in local database systems. These include:

Long-running transactions. Due to their long-running nature, MDBS transactions are highly resource-intensive. In the model of an *electronical software supermarket*, a transaction is composed of subtasks that can be bought by the systems of autonomous institutions in the network; the transaction provides support for the *coordination* of those services. An MDBS transaction must pay for the services it has used. Since the failure of one service must not cause the abortion of the entire MDBS transaction, alternative solution pathways must be available. Function replication, as suggested in the Flex Transaction model [11], suggests that there exist other systems in the network providing an equivalent service. Many other advanced models [11] incorporate the possibility of the specification of alternatives for failed components.

Early commitment. The lenghty holding of locks by an MDBS transaction at one local system may be a severe violation of local autonomy. In recognition of this, sev-

eral methods have proposed the relaxation the isolation property [1] of transactions, permitting every view of intermediate effects before the MDBS transaction commits. Transactions supporting cooperative work, such as cooperative design processes, then become possible.

To facilitate this relaxation, the concept of *semantic compensation* has been introduced. Atomicity requires that either all or none of the operations of a global transaction be performed. If side-effects can become visible early, other subtransactions may base their computations on this information. While an undo in the classical sense thus precluded, semantic compensation may be possible. Compensation, as originally proposed in [16], [14], is part of many advanced transaction models [11].

Environmental changes. The environment of an MDBS may change frequently: new local systems to be integrated may be added, the restrictions placed on local systems may change, an alternative global correctness criterion might appear more useful, or an application may require a different global transaction model. Given this fludity in models and criteria, the MDBS transaction manager must be flexible and allow dynamic adjustment to satisfy changing requirements.

Advanced transaction models have proposed various approaches to the above problems. Many of them extend or relax classical transaction properties. We will summarize some of the principal ideas of these new models and in particular emphasize the features of the Flex Transaction model [10], [24] in more detail.

An advanced transaction model requires a convenient notation with which to specify transactions. This language should be declarative, permitting the user to specify global application semantics possibly and to partipate in interactive communication (e.g., confirming the performance of certain tasks). A specification formulated in such a language has the additional advantage of being runnable. This language should be powerful enough to allow the modeling of *all possible* control and data flow patterns. It should not be limited to a set of predefined functions but should instead be based on an open framework like the ACTA (actions) system described in [7]. Moreover, the language should hide from the user all other aspects related to coordination, including communication reliability, implementation of concurrency, and recovery after system failures. We believe that this goal can be best achieved by a general-purpose, computationally complete programming language.

This paper is structured as follows. In Section 2, we derive the requirements for an MDBS transaction language by analyzing recent transaction models. In Section 3, we present the VPL (Vienna Parallel Logic) language and, in Section 4, we demonstrate, with several examples, how VPL can serve as the specification language for different kinds of transactional work flows. In Section 5, we demonstrate that transaction processing in VPL encompasses much more than the specification of the data and control flow between activities. For example, communication reliability is ensured and a well-defined recovery behavior after system failures is provided.

2. Specification Potential of MDBS Transactions

In this section, we analyze the information-carrying potential of an MDBS transaction as the first step toward defining useful linguistic constructs for an MDBS transaction specification language. We separate those properties of a global task that must be specified explicitly (including the subtransactions comprising a global transaction or the preferred subtransaction among alternatives) from those issues that can be hidden from the user. The latter include the physical distribution of services (modeled for the user by the name and location of a service), reliable communication protocols, and global correctness. While a language possessing many advanced features is more likely to satisfy all our requirements, we also hope to arrive at a straight forward structure which provides the essential features in a highly declarative way.

In the following subsection, we present a list of the most significant features offered by the advanced transaction models. This list is not comprehensive, since it is strongly constrained by the local systems to be integrated by a global transaction. The new requirements for transaction processing to be posed by future application domains cannot now be fully envisioned.

2.1. Alternatives

In a heterogeneous environment, partial failures may occur as single components suffer from independent failures. Moreover, in accordance with connection autonomy, systems may refuse connection over an unpredictable time period. As stated above, a global transaction is by nature a persistent activity for which abortion is undesirable. Therefore, in contrast to the approach taken by many classical transaction managers, the possibility of a rollback and re-start must be minimized if the transaction cannot accomplish its goal (this may occur if serializability is violated or a deadlock has arisen). This stipulation implies either that global concurrency control must be pessimistic or that the serializability requirement must be relaxed. Moreover, the global scheduler must be aware of services that can replace a failing subtransaction (*function replication*). Function replication provides a means for software fault-tolerance to be controlled at the application level.

The specification of *alternatives* can be made when the MDBS is installed and can later be refined. For example, information about alternatives can be stored in the integration directory along with information on logical data integration (e.g., semantic relations [20]). However, in many instances, the definition of alternatives is tightly bound to the semantics of the global transaction and must therefore be part of its specification.

Alternatives can be specified along several lines:

1 of n. In the simplest method, a task is defined and a number of servers are designated as capable of accomplishing this task. Preferences and constraints among these servers can also be stated (see Section 2.2), constraining the execution parallelism. "1 task of n tasks" has a declarative reading "task_1 OR task_2 OR ... OR task_n," with

OR denoting either sequential or parallel execution or an unspecified concurrency which depends on given preferences and constraints.

k of n. A task consists of k subtasks. A set of n servers ($k \leq n$) exists with the property that each server can perform one subtask. The task is fulfilled if all k sub-tasks have been performed by k different servers. Here, we assume that each subtransaction can be started only once; variants in which one server performs multiple subtasks are discussed in Section 2.6.

Selection of arbitrary patterns. In this most general case, n servers offer n different services. The goal of the global task is specified as a set of subsets of the n services. The global task has been accomplished when one such subset has been accomplished.

2.1.1. Alternatives in Flex Transactions

The Flex Transaction model supports nondeterministic specifications through the definition of multiple *commit sets*. Every commit set represents an acceptable state; preferences between these commit sets cannot be defined. Thus, an execution model for a Flex transaction is designed to accomplish any of these possible commit sets in a nondeterministic manner. This is similar to the nondeterministic selection of one from among several potential guarded clauses [8]; there, the first clause to complete its guard is usually selected. Similarly, in the Flex execution model [23], all choices are left open for as long as possible, but the first commit set to be accomplished is then given preference. The Flex model is a comprehensive framework that can encompass the selection of any alternative, as shown in these examples.

Example 1 *(1 of n)*
Let t_{klm}, t_{aua}, t_{twa}, and t_{delta} be airline reservation transactions. The set of commit sets $\{\{t_{klm},\},\{t_{aua},\},\{t_{twa},\},\{t_{delta},\}\}$ specifies the selection of one airline transaction from among these four. Thus, the goal of the Flex Transaction is achieved if a flight can be booked at any one of the airline databases.

Example 2 *(k of n)*
Let t_{opera}, $t_{concert}$, and $t_{theatre}$ be transactions that represent evening activities in Vienna. The task of the Flex Transaction is to reserve tickets for two evenings activites without duplication. The set of commit sets is therefore $\{\{t_{opera},t_{concert}\}, \{t_{opera},t_{theatre}\}, \{t_{concert},t_{theatre}\}\}$, selecting two of three events.

Example 3 *(pattern)*
A trip from Vienna to Tunis can be made either by booking a flight or by booking travel by train and by ship. The set of commit sets is therefore $\{\{t_{flight}\}, \{t_{train}, t_{ship}\}\}$. This Flex Transaction specifies the selection of two patterns out of three transactions.

The Flex model allows the syntactic specification of subsuming patterns. If the commit set C = $\{t_{train}, t_{ship}, t_{car}\}$ is added to Example 3, C subsumes the commit set $\{t_{train}, t_{ship}\}$. In this specification, t_{car} is a *non-vital* transaction, as defined in transactions. In

the Flex execution model proposed in, [23] we have simply excluded such specifications and thus would ignore commit set C. Non-vital transactions are also discussed in [4] and [28].

2.2. *Execution State Dependencies*

During its lifetime, a transaction may pass through several execution states, which are listed in Table 1. After state P is reached, a decision to commit or abort the transaction is made by the global transaction manager.

Table 1. Transaction execution states

N	not executing (not yet begun)
E	executing (begun)
P	prepared (if two-phase commit is provided)
S	succeeded (committed)
F	failed (aborted)

If the local scheduler of a given transaction supports a visible two-phase commit protocol [1], the transactions may be as follows:

$$N \rightarrow E \rightarrow P \rightarrow S$$

$$N \rightarrow E \rightarrow P \rightarrow F$$

If the local scheduler of a transaction does not support two-phase commit, so that the local transaction commits or fails immediately, the transitions may be as follows:

$$N \rightarrow E \rightarrow S$$

$$N \rightarrow E \rightarrow F$$

Dependencies may be placed on any of these execution states. The transition of a transaction to its next state may thus be dependent on the execution state of one of more other transactions of the same global transaction. Execution state dependencies constrain the potential parallelism of a specification and can be viewed as concurrency control statements.

Example 4 *(Concurrency Control)*
Let t_{flight} and t_{hotel} be flight and room reservation transactions, respectively. Let a dependency be specified by which t_{hotel} must not be started before t_{flight} has succeeded (i.e., the transition $N \rightarrow E$ of t_{hotel} depends on the success of t_{flight}). This dependency constrains the execution of t_{hotel} to be sequentially implemented after the successfully completed execution of t_{flight}.

Similarly, we may add two transactions t_{avis} and t_{hertz} to reserve a rental car, and state that the start of t_{hertz} depends on the failure of t_{avis}; this is another example of sequential execution.

In summary, concurrency allows the specification of:

- **sequential** or
- **parallel** or
- **neutral** execution.

While parallel execution requires a *fair* parallel execution of the transactions, neutral execution includes the possibility that the transactions may run either in parallel or in any sequential order. Neutral specifications are more declarative than are their parallel and sequential counterparts as they leave the concurrency control specification open. Depending on hardware and network capabilities, the scheduler (compiler/interpreter) may select between parallel or sequential execution.

2.2.1. Execution State Dependencies in Flex Transactions

The Flex Transaction model supports *internal dependencies* that allow the start of sub-transactions to be dependent on the success or failure of one or more other transactions (positive/negative dependencies). Recursive dependencies are forbidden in a correct specification. As Flex supports nondeterministic specifications (see Section 2.1), not all dependencies are relevant for all possible commit sets [23], [19]. We use the relation "$\prec_S$" to denote a success dependency between two transactions t_i and t_j; $t_i \prec_S t_j$ states that t_j must not be started until t_i has succeeded. A success dependency $t_i \prec_S t_j$ need not be tested if t_i or t_j will not be part of the final solution; i.e., there is no possible commit set to which both, t_i and t_j belong. In general, a success dependency ($t_i \prec_S t_j$) is relevant for a certain commit set C if and only if $t_i \in C$ and $t_j \in C$. A failure dependency is denoted by $t_i \prec_F t_j$ and states that the start of t_j depends on the failure of t_i. A failure dependency $t_i \prec_F t_j$ is relevant for a commit set C if and only if $t_j \in C$ and $t_i \notin C$.

Example 5 *(Relevance of Dependencies in Flex Transactions)*
Let us assume the same set of commit sets as in Example 3. A success dependency exists between t_{train} and t_{ship} ($t_{train} \prec_S t_{ship}$) by which the ship must not be booked unless a train ticket is booked. If there are no further dependencies specified, the flight reservation transaction can run in parallel. For the flight transaction, the success dependency $t_{train} \prec_S t_{ship}$ is irrelevant, because both t_{train} and t_{ship} are extraneous to the commit set to which t_{flight} contributes.

Example 6 *(Contradictory Dependencies (a))*
Using once again the commit sets of Example 3, let us now assume that there exists a failure dependency $t_{train} \prec_F t_{ship}$. For the commit set $\{t_{flight}\}$, this dependency is irrelevant, but it introduces a contradiction for the commit set $\{t_{train}, t_{ship}\}$; both transactions are needed for this commit set, but the dependency indicates that t_{train} must fail in order to allow the start of t_{ship}.

Syntactically, the Flex model does not restrict the specification of such dependencies, but the execution model for Flex Transactions handles "contradictory" specifications correctly in that it ignores irrelevant dependencies. If required, adequate warnings can

be provided. Another example of contradictory dependencies, with $t_i \prec_F t_j$ and $t_i \prec_S t_j$, is as follows:

Example 7 *(Contradictory Dependencies (b))*
Let us assume a Flex Transaction consisting of two airline transactions t_{klm} and t_{twa} and the hotel reservation transaction t_{hilton}. The set of commit sets consists of $C_1 = \{t_{klm}, t_{hilton}\}$ and $C_2 = \{t_{twa}, t_{hilton}\}$. The internal dependencies are $t_{klm} \prec_S t_{hilton}$ and $t_{klm} \prec_F t_{hilton}$.

According to our definition of relevant dependencies, $t_{klm} \prec_S t_{hilton}$ is relevant only for commit set C_1, and $t_{klm} \prec_F t_{hilton}$ is relevant only for commit set C_2. Thus, the semantics of this Flex Transaction specification are that t_{klm} must be executed before t_{hilton} and t_{twa} can be executed. If t_{klm} succeeds, t_{hilton} is executed. If t_{klm} fails, t_{twa} and t_{hilton} are executed in parallel.

The same situation could be represented in a more declarative manner with the following dependencies: $t_{klm} \prec_S t_{hilton}$ and $t_{klm} \prec_F t_{twa}$.

If failure dependencies are used in combination with function replication, they may be understood as preferences among alternatives, as shown in the following example.

Example 8 *(Preferences)*
Let us add to Example 1 the following failure dependencies: $t_{aua} \prec_F t_{klm}$, $t_{klm} \prec_F t_{delta}$, and $t_{delta} \prec_F t_{twa}$. This specifies the following preference ordering: try t_{aua} first, then t_{klm}, then t_{delta}, then t_{wa}, and finally give up. The addition of these failure dependencies transforms the original specification of parallel function replication into sequential function replication.

Internal dependencies are defined over the domain of those transactions which make up a Flex Transaction. They have the following properties:

(A) $t_i \not\prec_S t_i$.

(B) $t_i \not\prec_F t_i$.

(C) If $t_i \prec_S t_j$ then $t_j \not\prec_S t_i$.

(D) If $t_i \prec_F t_j$ then $t_j \not\prec_F t_i$.

(E) If $t_i \prec_S t_j$ and $t_j \prec_S t_k$ then $t_i \prec_S t_k$.

(F) If $t_i \prec_F t_j$ and $t_j \prec_F t_k$ then $t_i \prec_F t_k$.

(G) If $t_i \prec_F t_j$ and $t_j \prec_S t_k$ then $t_i \prec_F t_k$.

(H) If $t_i \prec_S t_j$ and $t_j \prec_F t_k$ then $t_i \prec_S t_k$.

As stated previously, recursive specifications are automatically excluded. The Flex Transaction specification must be completely defined, in that all dependencies must be explicitly stated. For example, in Example 8, the dependencies $t_{aua} \prec_F t_{delta}$, $t_{aua} \prec_F t_{twa}$, and $t_{klm} \prec_F t_{twa}$ must be added to produce a correct specification.

The above discussion indicates that the formulation of Flex Transactions by means of internal dependencies is not trivial. The Flex Transaction model is a comprehensive

framework that permits the formulation all types of concurrency control patterns. However, specifications of large transactions would become quite cumbersome if the original notation is used.

Table 2. Dependencies in ACTA

dependency	relation	description
commit	t_i $\mathcal{CD}$ t_j	if both transactions commit, then the commitment of t_i must precede that of t_j
strong-commit	t_i $\mathcal{SCD}$ t_j	if t_i commits, then t_j commits
abort	t_i $\mathcal{AD}$ t_j	if t_i aborts, then t_j aborts
weak-abort	t_i $\mathcal{WD}$ t_j	if t_i aborts and t_j has not yet committed, then t_j aborts
termination	t_i $\mathcal{TD}$ t_j	t_j cannot commit or abort until t_i either commits or aborts
exclusion	t_i $\mathcal{ED}$ t_j	if t_i commits and t_j has begun executing, then t_j aborts
force-commit-on-abort	t_i $\mathcal{CMD}$ t_j	if t_i aborts, then t_j commits
begin	t_i $\mathcal{BD}$ t_j	t_j cannot begin executing until t_i has begun
serial	t_i $\mathcal{SD}$ t_j	t_j cannot begin executing until t_i either commits or aborts
begin-on-commit	t_i $\mathcal{BCD}$ t_j	t_j cannot begin executing until t_i commits
begin-on-abort	t_i $\mathcal{BAD}$ t_j	t_j cannot begin executing until t_i aborts
weak-begin-on-commit	t_i $\mathcal{WCD}$ t_j	if t_i commits, then t_j can begin executing

2.2.2. *Execution State Dependencies in ACTA*

The ACTA framework [7] allows general execution state dependencies to be specified as binary relations between two transactions t_i and t_j. It is assumed that a history exists which indicates the relative sequence of events, each event being a state transition of a transaction. For example, the dependencies that can be modeled using the ACTA open framework are shown in Table 2.

Table 3 shows the established correspondences.

Table 3. Correspondences

t_i $\mathcal{BCD}$ t_j	$\equiv$	$t_i \prec_S t_j$
t_i $\mathcal{BAD}$ t_j	$\equiv$	$t_i \prec_F t_j$

2.3. *Commit Granularity and Compensation*

In Section 1, we explored the rationale for the relaxation of the isolation property of classical transactions in many advanced transaction models and the consequent need for semantic compensation. In the environment, the control of transaction commitment becomes highly problematical. We will first explore the possibility of specifying the granulanity of commitment as an alternative to the immediate commitment of every transaction.

2.3.1. *Commitment in Nested Transactions*

Classical nested transactions [25] do not relax the isolation property. A transaction depends on all of its subtransactions, and the commitment of a subtransaction is delayed until the global (highest-level) transaction commits. Thus, the granularity of commitment is implicitly at the level of the entire global transaction.

This idea of nested transaction concept has been incorporated into advanced transaction models, necessitating refinement of the notion of commitment granularity in that context.

2.3.2. *Compensation in Sagas*

In the Saga model, a global transaction (or Saga) consists of several transactions $t_1,\ldots,t_n$. Each t_i $(i = 1,\ldots,n)$ is accompanied by a compensating action c_i. The commitment granularity is implicit in this model, in that every t_i commits immediately. t_i must not be started unless t_{i-1} has committed. The Saga succeeds if t_n succeeds. If t_{i+1} fails, the compensating actions $c_i,c_{i-1},\ldots,c_1$ are executed.

2.3.3. *Compensation and Commitment in Flex Transactions*

In the Flex Transaction model, transactions are specified as *compensatable* or *noncompensatable*. Transactions of the former type commit immediately and must be coupled with compensatory actions. Noncompensatable transactions are required to support a two-phase commit protocol in which their commitment can be triggered by the commitment of the global transaction.

The distinguishing feature of Flex Transactions is the inclusion of transactions of both types in nested structures. A Flex Transaction may include not only subtransactions at local systems but may also be composed of other Flex Transactions. The commitment granularity is explicitly determined by the transaction type. A transaction of the noncompensatable type delegates its commitment to its caller (the innermost enclosing transaction).

If a transaction is of the compensatable type, it commits immediately. This commitment implies the commit of all nested noncompensatable subtransactions .

A compensatable subtransaction that has committed is compensated if either:

- the Flex Transaction fails (e.g., there are no additional commit sets that can be fulfilled), or
- the Flex Transaction succeeds with a commit set C so that $t \notin C$, or
- the Flex Transaction is aborted (e.g., by an external signal, or by an encosing Flex Transaction).

Additionally, no enclosing transaction of t may have committed, preventing cascading compensations. In terms of software composition, these compensating semantics are

structured with a larger module (transaction) which is responsible for the handling of its composite the sub-modules.

Example 9 *(Noncascading Compensation)*
Let us assume a compensatable Flex Transaction T_{London} that arranges a trip to London consisting of flight, hotel, and car reservations and a couple of tickets for evening events. Assume that airline transactions are compensatable: a ticket can be returned without incurring extra costs. Although a user may find no tickets are to be available, a seat may later open up because another flight reservation has been compensated. All other transactions are noncompensatable.

Let T_{London} be part of another Flex Transaction specification $T_{holiday}$ that books a package trip to London for a couple and at the same time books a sailing course for their son (T_{sea}).

Let us now assume that T_{London} commits and T_{sea} cannot be fulfilled; T_{London} is then compensated. This is accomplished by calling the compensate action of the T_{London} package, thus compensating, the originally noncompensatable hotel, car, and evening reservation transactions.

2.3.4. Representation of Compensation and Commitment in ACTA

In the ACTA framework, the granularity of commitment can be modeled by means of *delegation*, in that a transaction may delegate the responsibility of commitment to another transaction. The ACTA model can be used to control a variety of levels of commitment granularity. In [7], Sagas are formalized through the medium of ACTA.

2.4. Data Dependencies

Dependencies may exist among the data components of a transaction. Some transaction models, including ACTA, assume that a transaction can be decomposed into read/write operations, information which is then used to guide global schedulers. The subactivities of a global transaction are termed *methods*. An MDBS system which accesses a server or local database system through a local user interface may not necessarily encounter a read/write operation interface. More advanced applications such as multi-media systems, computer aided design systems, or cooperative workgroups, typically provide high-level functions. When the semantics of a method are known, data dependencies can be derived and used for global concurrency control.

More important are data dependencies that refer to output data of a transaction. Such dependencies may dynamically refer to the *quality* of a produced solution. For example, a user may state that, if a flight costs less than a specified amount, a more expensive hotel room can be booked. The user may constrain the total price of a trip, encouraging alternatives to be tried. These examples involve the definition of the *semantic failure* of a transaction, or situations in which a solution does not satisfy certain conditions.

Current advanced transaction models do not offer a satisfactory solution to the issue of data dependencies. The Flex Transaction model and Sagas provide no support for data dependencies.

2.5. *User Interaction*

The above examples indicate the importance of user input in defining the semantics of the global transaction. Such preferences as a short layover time versus a higher fare must be formulated by the user. Such information varies with each transaction, requiring a declarative language that can be easily used by an intermediary such as a travel agent. This situation is essentially different from a classical database system which attempts to mask all control information from the user.

The MDBS user formulates queries over a larger, more complicated, and heterogeneous set of data. Thus, more responsibility is required and a complete transparency of all underlying heterogeneities can never be entirely achieved. We believe that dynamic and ad-hoc approaches are needed, which can later be refined toward a tighter and more static integration.

It is clearly impossible to envision all possible situations and queries which may arise in a public network consisting of hundreds or thousands of database systems. The MDBS user must therefore be equipped with a tool that is not restricted to a certain set of vendor-defined functions. An MDBS user who is willing to learn a single powerful, high-level, general-purpose tool will have achieved freedom from software support teams or the need to await more sophisticated versions of specialized MDBS transaction software.

An analogy may be made with the growing role of CAD systems. Since many designers fear becoming dependent on a single office CAD expert, some offices have avoided the purchase of any CAD system. Similarly, travel agents will chafe at the restrictions imposed by the pre-defined functions of their transaction software by their dependency on outside support, particularly when these factors prevent the formulation of certain client requests.

Users will clearly be involved in non-static MDBS transaction specifications, adding interactivity to the requirements of any declarative transaction specification language. Throughout the execution of a long-lived transaction, such as those in a cooperative workgroup, user input may be necessary. Some subtasks of a global transaction, such as making a phone call or sending a fax, may be external events that cannot be performed by a computer but must be performed or acknowledged by the user.

2.6. *Advanced Control Mechanisms*

Several mechanisms are available to achieve more sophisticated control of transactions. These include:

External constraints. The transition of a transaction to another execution state can be made dependent on environmental or *external constraints*. In the Flex Transaction

model, the start of a transaction can be defined as dependent on time constraints. For example, booking an opera ticket can be constrained to be performed during office hours.

Other constraints that can be imposed include resource availabliltiy, current server load (devices from management information about the network load), the state of a permanent object, or the recurrence of certain events.

The imposition of external constraints may either be immediate or may be involved when a specified event occurs (*delayed decision*).

Timeout. Timeouts are essential to guarantee the correctness of a global transaction. Although the generation of timeouts is primarily under the control of the transaction execution model, they may also be specified as part of the semantics of the global transaction. For example, in the travel example provided previously, if a flight reservation cannot be made prior to a certain date, the flight transaction should be abandoned.

Retry condition. A retry condition is a method of handling failures. While function replication substitutes a failed component with another, a retry condition retries the same transaction again. A retry condition may involve a counter or other constraints and can be associated with a time-interval.

For example, at a weekly time-interval, a client may check with an airline on the availability of inexpensive seats that have opened up due to cancellations. The retry condition ensures that no other flight has been booked by the client and that the travel date is not overshot.

Repetition. A global transaction may require several executions of the same subtransaction. Let us assume that a travel club wishes to book a flight for all n members—on the same airline.

If a method that represents the flight reservation cannot be parameterized to simultaneously handle n tickets, it must instead be possible to generate several instances of the same transaction. The original Flex Transaction model employed retrial and repetition. Later work on this model, however, restricts each subtransaction to a single use. Repetition can therefore only be simulated by artificially renaming transactions; in our example, $t_1,\ldots,t_n$ would all denote the same subtransaction. This somewhat clumsy approach merits further investigation.

Dynamic interrupts. The global transaction manager automatically aborts unneeded sub-transactions. There is clearly a need for user-controlled *interactive interrupts* which permit the aborting of a subtransaction or the global transaction at any time. More advanced signals like *pause* and *continue* may also be constructed, and it may be important to *migrate* an activity to another site. These latter features are primarily useful for long-running transactions.

Current state. Users may wish to determine the current state of a transaction and to get detailed information about the execution states of all subtransactions and their results

to date. In the travel agency used above, the client may wish to know what parts of the trip have already been booked and at what price. In a workflow management system with automatized office control flow, a user may wish to learn the actions triggered by a fax she sent.

Dynamic changes. If a user orders a complicated trip-booking transaction composed of many subtransactions which takes several days to implement, it is possible that, within this time, the client requirements may change, necessitating ad-hoc alterations in the transaction specification. Such a retrospective change of a running process is clearly a difficult task.

3. A Parallel Language Based on Logic

The advanced transaction features discussed above can either be supported by a specialized operating system layer or by a distributed and parallel programming language that provides concurrency, communication, and synchronization. Moreover, the maintenance of persistent objects and a well-defined behavior in case of failure are essential for such a language. An instructive comparison may be made with the implementation requirements for the ConTract model, a model for defining and controlling long-lived, complex computations which is decribed in [29].

Traditionally, programming support has been provided by operating systems. However, sockets, RPC (remote procedure call), and other related operating system libraries are not well suited to MDBS transaction specification. Even the most advanced libraries which provide reliable group-based communication, such as the ISIS programming package described in [2], are insufficient to express a complex MDBS transaction.

Recently, specialized workflow languages like IPL [5] (InterBase Programming Language) have been developed. IPL has been developed to implement Flex Transactions and provides graphical user interfaces. The limitation of such special-purpose languages lies in their close relationship with a single model, which is most often within the imperative programming paradigm.

A rule-base language is well suited to approach many of the issues explored in the previous section. Dependencies can be formulated quite naturally as rules, which can be stored in an integration directory. Intelligent integration rules can be supported, and tight integration can be maintained. Knowledge can be automatically derived from existing rules. Given these merits, we have therefore directed our efforts toward the development of a rule-based language for the specification and control of the execution of multidatabase transactions. In this paper, we focus only on transaction control. A rule-based language also offers advantages for the representation of semantic data integration [20], since first-order logic naturally suits the representation of SQL statements.

3.1. *History of VPL*

The design of VPL (Vienna Parallel Logic) was principally motivated by the need to represent transaction control in Prolog. Prolog was at that time employed for semantic

data integration (a problem independent of physical distribution), and we subsequently recognized that explicit language constructs were needed to specify transaction control.

Our investigations concluded that languages from the family of concurrent logic languages [26] are unsuitable for MDBS transaction specification. Their draw backs include:

- They forbid backtracking; only a single solution can be computed. The formulation of alternatives is therefore precluded.
- Their failure behavior is unacceptable; if one process fails, the whole system fails.
- They do not provide a reliable communication mechanism; a course of action in case of communication and systems failures is not specified.
- The commitment granularity is limited to a single clause, making delegation impossible.
- They do not allow the explicit spawning of a process at a remote site; a request for a service at a particular server cannot be specified.

Concurrent logic languages do incorporate many attractive features, including a high-level, shared-data-based communication via logic variables and declarative constructs to model concurrency. However, they are unsuitable in open environments where partial failures may occur.

Influenced by concepts embodied in the Flex Transaction model, we have developed a *coordination paradigm* [18] that includes concurrency, reliable communication, transactions, and object-orientation[1]. This toolkit can be incorporated into any existing programming language. Our extension of the logic-based programming language Prolog are termed Prolog&Co and VPL. Prolog&Co adds the toolkit to Prolog in form of a set of built-in functions, whereas VPL represents a smooth embedding of the toolkit into the Prolog syntax. Another difference to Prolog&Co is that VPL is multi-threaded. Similarly, C extended by coordination results in C&Co [13], [12]. As the toolkit adopts many concepts from the Flex Transaction model, any representative of the resulting class of coordination languages is particularly well suited to the representation of Flex Transactions.

In the following section we will discuss the VPL language in more detail and illustrate its application to transaction specification and execution. Every specification given in terms of VPL is runnable and can be executed as is by a VPL runtime machine. Thus the examples provided below can be directly executed on our prototype VPL engine.

3.2. *A Brief Language Overview*

VPL is a superset of sequential Prolog and is described in [21] in more detail. The extensions in VPL include:

3.2.1. *Concurrency*

Concurrent language constructs serve to model parallelism explicitly. Sequential AND/OR constructs of Prolog can be extended naturally to be parallel or neutral.

AND operators are described in Table 4. G denotes a goal (in terms of transactions, this is a subtransaction or a subactivity). G succeeds if both G_1 and G_2 succeed. Each composition of goals by means of an AND operator results in a new goal. The failure of a goal results in *backwards execution.*

Table 4. AND operators in VPL

goal G	**semantics of G**
G_1 & G_2	sequential AND: the start of G_2 depends on the success of G_1
G_1 && G_2	parallel AND: G_1 and G_2 are started in parallel
G_1 and G_2	neutral AND: G_1 and G_2 can be executed in any order

Prolog (and VPL) is tuple-oriented. Solutions are tried one after another, with every solution addressing a single tuple rather than a set. If alternative routes to the fulfillment of a goal are specified, an alternative solution path can be explored. A general search over all possible solutions is implemented by a *backtracking* procedure which, unlike exhaustive search, can be implemented efficiently on traditional computer systems.

A clause has the form "H ← G.". A procedure consists of several clauses with the same head functor. All clauses of a procedure must have the same rule-operator ← (":–", "::–", or "<–"). A procedure defines the method of achieving a goal. If n clauses with the same head are specified, there exist n alternative ways of fulfilling the goal. The first successful alternative is adopted. The ordering in which of alternatives are tried is explained in Table 5. If no "|" (*commit operator*) appears in G, another clause can be selected on backtracking.

Table 5. OR operators in VPL

clause C_i	**semantics of C_i's procedure**
H :- G.	sequential OR: C_{i+1} is started only after C_i has failed
H ::- G.	parallell OR: all C_i are started in parallel
H <- G.	neutral OR: the C_i can be executed in any order

AND/OR create concurrent processes which are controlled by the local VPL system. To create a process at a VPL system at a remote site, the primitive "process(Where, Goal, PID, Type)" is used. "Where" specifies a coordination system (e.g., VPL or C&Co) at a certain site; e.g., vpl@tuwien.ac.at. "Where" can also be defined either as 'LOCAL,' meaning that the process will run as another thread in the local VPL system, or as 'REMOTE', meaning that the process can be sent to any reachable VPL system on the Internet (automatic load balancing). "Goal" represents the task to be executed by

the process. "Type" is either 'DEP' (dependent) or 'INDEP' (independent). "PID," or process identification, is a unique identifier of the process.

The process primitive starts the process and succeeds immediately. The result of a process can be checked by testing its PID. The commit "|" of a 'DEP' process (see transactions below) is delegated to its caller.

A transaction (see Section 3.2.4 below) depends on processes of type 'DEP' and implicitly waits until all PIDs are identified as 'SUCCEEDED.' Processes of type 'INDEP' are similar to *vital* transactions, and the caller is not dependent on their termination state.

If a process of type 'INDEP' is not terminated when a system failure occurs, it will be automatically restarted with the same arguments (i.e., the same *image*) after the system has recovered. If a process is started a second time[2], VPL tests whether the process has already terminated or is still running, in which case it is re-started with its previous image. All 'INDEP' processes are started automatically, and all 'DEP' processes are re-started by recovered 'INDEP' processes. If, on re-execution, an 'INDEP' process is encountered that has already been automatically recovered, the second re-start from the VPL program is ignored.

The primitive "signal" sends a signal ('ABORT,' 'PAUSE,' 'CONTINUE,' 'MIGRATE') to a process that is uniquely identified by its PID.

3.2.2. *Communication*

Communication is accomplished via shared logic variables. In VPL, both the ordinary Prolog variables and additional communication variables are supported. The bindings (assigned values) of the former are undone on backtracking (like in Prolog), while a value written into a communication variable is permanent. Communication variables are created with the operator "#" and are persistent.

Communication variables can be shared between parallel or distributed processes (by passing them in the arguments of a goal) and form a unique mechanism for inter- and intra-process communications. All participating processes maintain an unchanging and shared view of an object. Communication variables are incorporated in a distributed environment through replication strategies based on primary copy migration [22]. They offer the advantages of resiliency to system failures and reliability of communication. Communication variables can contain other communication variables as subcomponents which are automatically shared. They can thus be used to construct a variety of communication structures, including streams.

3.2.3. *Implicit Synchronization*

Unification ("=") in Prolog between two terms can be considered as a symmetric assignment operation that tries to equalize both terms, possibly by binding variables on both sides. The existing Prolog unification mechanism may encounter a communication variable which must be bound to produce this state of equality. In this case, the unification mechanism suspends until another concurrent process writes a value to that

variable. The ordinary unification mechanism is prohibited from binding a communication variable. Note that the selection of a clause (*goal-head unification*) employs also "=" unification and thus can cause implicit synchronization.

3.2.4. Transactions

The "|" (*commit*) operator determines language *transactions* (versus the database transactions addressed in Section 3.3) and can appear in goals at any point at which an AND operator can appear. It can also appear directly after the rule-operator and at the end of the clause. "|" has the declarative reading "AND." "|" acts to symmetrically cut alternatives in sequential, parallel and neutral procedures and also controls commitment granularity.

A communication variable is bound, or assigned a value, by a new unification operator "=#=" (*full unification*). Values assigned in this way become visible only when the next "|" operator is executed (in one atomic step). This step renders the value of the communication variable globally visible to all processes that share it. The atomic writing of a group of communication variables is thus made possible. The statement that a communication variable is written means more precisely that a value is issued via "=#=" and then committed.

The predicate "cvar(X)" serves to test whether its argument is still an undefined communication variable; the test is assured on transaction commit.

Within a transaction, compensating actions can be defined by means of the predicate "compensate(Action)." They are collected until the next "|" where they are enabled but not executed.

If backtracking occurs, a committed transaction is not rolled back over its individual goals, but instead all its compensating actions are executed. As in the Flex Transaction model, outermost transactions are responsible for the compensation of their subtransactions; there are no cascading compensations.

The predicate "prepare(Action)" is designed in analogy to the prepared phase of classical transactions. The prepared phase within VPL acts semantically to define an action that will be executed when it is clear that the VPL transaction can succeed. It defines a post-condition for a transaction that is executed on commitment.

3.3. Local System Interface

Interfaces to local systems are not an intrinsic VPL function but can readily be implemented using VPL. The interface to a local system to be integrated must be conceptually simple to permit systems to be easily exchanged without modifying the transaction specification. Such an interface is similar to that of the ConTract model [29], which differentiates between *scripts* and *steps*. The latter formulate actions to be taken at the local system and are thus defined by the specifications of the interface to that system. Scripts program the composition of steps and thus correspond to what we call "explicit MDBS transaction specifications."

Our proposed interfaces follow a procedure termed "remote_call(Where, Query)." Query may have the form "query(In, Out, Err)" if the local system does not understand Prolog; otherwise, it can directly contain a goal. A remote_call is responsible for the concurrency control protocol in the MDBS. In [6], we describe remote_calls under a variety of assumptions about local systems. Since a remote_call consists of only about 15 lines of VPL code, VPL itself can serve to implement the communication protocols of the MDBS. In this paper, we assume that these protocols are already supported in a library and need not be specified by the user. The user may select the appropriate remote_call; remote_call_{NC} calls a noncompensatable transaction, while remote_call_{C} calls a compensatable transaction. Remote_call_{C} executes a "|" and thus commits the database transaction it calls. Within the committed language transaction, a compensating action is defined. If the remote_call_{C} is later aborted by its calling transaction, this compensating action is called automatically. In contrast, the commit of remote_call_{NC} is delegated to its caller; thus, many database subtransactions can be committed within one atomic step.

It is important to note that the *language commit* triggers commitments in local database systems, and the specification of a *language compensate action* can be a remote_call that represents the compensating action of the database transaction. All these actions are controlled by the VPL language.

3.4. *System Architecture*

At every participating site, a *coordination kernel* must be running as an operating system process, as shown in Figure 1. A coordination kernel implements reliable communication between processes. Every VPL or C&Co system communicates with its local coordination kernel to access a communication variable (also termed *communication object*). The VPL system itself provides the interface to a local database system. In other MDBS architectures, this is also called a "gateway process," "multidatabase interface," "remote system interface," or "local access manager."

In contrast to client-server based MDBS architectures, any VPL system can serve as the MDBS in the architecture proposed here. The shared data communication paradigm results in symmetric communication, and, if the necessary remote_call procedures are supported for every VPL system, there is no need for a single centralized MDBS.

4. Transaction Specification With VPL

Using explicit concurrency operators, within VPL, a large class of relevant transactions can be specified. More precisely, transactions that have a declarative reading can be formulated. Communication variables, permit the modeling of all types of control and data flow. In [23], we demonstrated the automatic mapping Flex Transaction into VPL programs. We also analyzed a relevant sub-class of Flex Transactions, termed *binary* Flex Transactions, that are directly modeled by sequential/parallel AND/OR.

In the ensuing sections, we will first illustrate declarative transaction control with selected examples (Section 4.1). Declarative transaction control permits specification

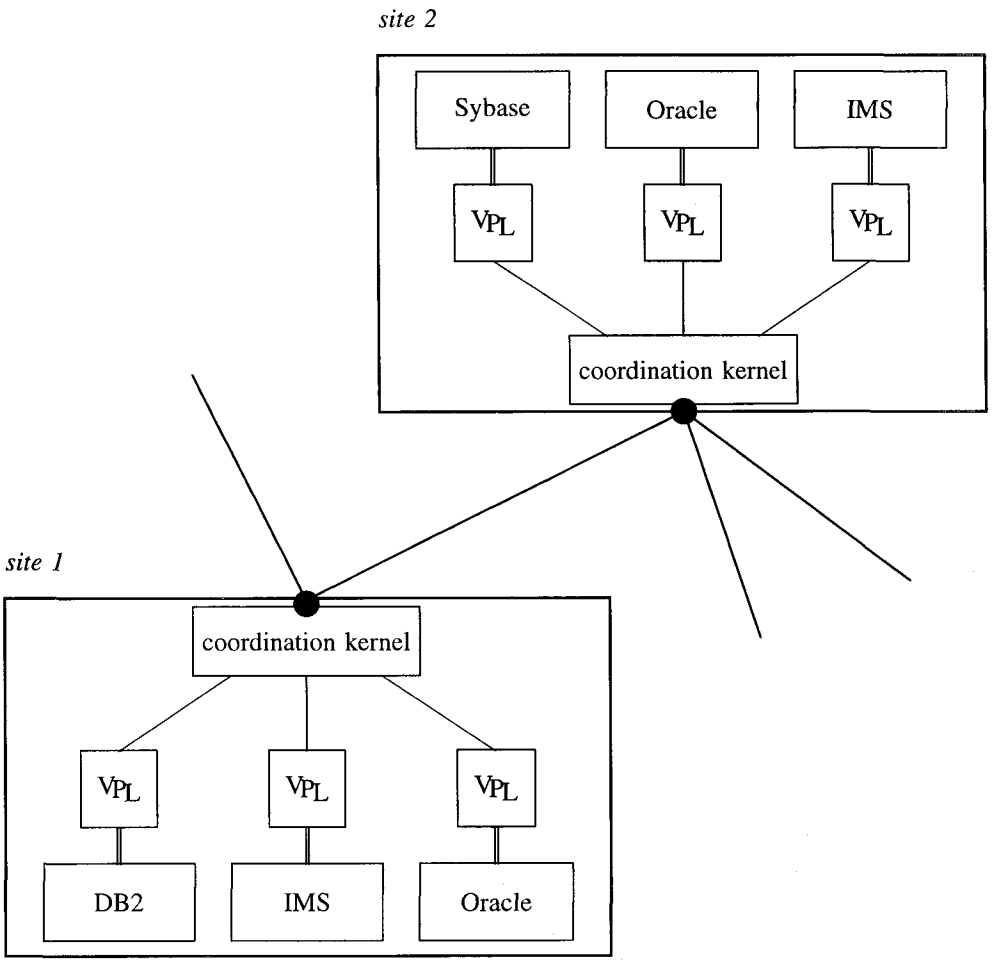

Figure 1. Systems Integration with the VPL Coordination Language

solely through the concurrency and transaction mechanisms of VPL. In Section 4.2, we provide examples involving communication variables

In the following, LTA_i denotes a local subtransaction at a local database system called from VPL. LTA_i is the goal "remote_call($LDBS_i$, query(t_i,Result,Error))," where $LDBS_i$ specifies the VPL system that serves as the interface to the local database system where t_i is executed. LDBS is the abbreviation for "local database system." A local subtransaction LTA_i must be started only once during the execution of its global transaction, denoted by gta. If the gta uses communication variables, we assume that they are passed as arguments (denoted simply by "gta(...)"); this is advantageous if the gta is recoverable (see Section 5.1).

4.1. Declarative Transaction Control

4.1.1. Alternatives

Alternatives are represented by defining several clauses for a procedure.

1 of n. Logical OR selects one of n alternatives. The rule-operator provides control over the trial of alternatives in parallel, sequence, or without specification.

Example 10 *(1 of n (Example 1 specified in VPL))*
In this example, no failure dependencies exist between t_{klm}, t_{aua}, t_{twa}, and t_{delta}. This situation can be specified as a global transaction gta_{10}:

gta_{10} <– LTA_{klm}.
gta_{10} <– LTA_{aua}.
gta_{10} <– LTA_{twa}.
gta_{10} <– LTA_{delta}.

k of n. The selection of k of n alternatives cannot be formulated declaratively without constraining the possible degree of parallelism or introducing artificial dependencies. "k of n" must be modeled with the assistance of communication variables; this will be illustrated in Section 4.2.

In the following example, we attempt to model the selection of two events out of three by artificially preferring a solution involving the procurement of opera tickets. As soon as an opera ticket can be booked, it is committed (this is a modification of the original problem formulation).

Example 11 *(k of n (modified Example 2 specified in VPL))*
gta_{11} :– LTA_{opera} | one_of($LTA_{theatre}$,$LTA_{concert}$).
gta_{11} :– $LTA_{theatre}$ | one_of(LTA_{opera},$LTA_{concert}$).
gta_{11} :– $LTA_{concert}$ | one_of(LTA_{opera},$LTA_{theatre}$).

one_of(LTA1, LTA2) <– LTA1.
one_of(LTA1, LTA2) <– LTA2.

Selection of arbitrary patterns. If patterns are not overlapping, they can be represented declaratively; otherwise, a similar problem may arise to that encountered when selecting k of n.

Example 12 *(pattern (Example 3 specified in VPL))*
gta_{12} <– LTA_{train} and LTA_{ship}.
gta_{12} <– LTA_{flight}.

4.1.2. Execution State Dependencies

A failure dependency can be modeled with sequential OR. Analogously, a success dependency constrains AND parallelism and requires sequential AND. The success or failure of a transaction t in VPL is directly reflected in the success or failure of the goal that calls t, thus avoiding the need for extensive state-testing.

Example 13 *(preferences (Example 8 in VPL))*
gta_{13} :– LTA_{aua}.
gta_{13} :– LTA_{klm}.
gta_{13} :– LTA_{delta}.
gta_{13} :– LTA_{twa}.

If some alternatives are to be tried in parallel and others in sequence, then the procedure must be split into several segments, as in this example.

Example 14 *(preferences (modified Example 1 in VPL))*
Let us assume that the following failure dependencies are given which indicate that European airlines are preferred: $t_{aua} \prec_F t_{delta}$, $t_{klm} \prec_F t_{delta}$, and $t_{aua} \prec_F t_{twa}$, $t_{klm} \prec_F t_{twa}$.

gta_{14} :– european_airline.
gta_{14} :– us_airline.

european_airline <– LTA_{aua}.
european_airline <– LTA_{klm}.

us_airline <– LTA_{delta}.
us_airline <– LTA_{twa}.

4.1.3. Commitment Granularity and Compensation

A desired commitment granularity can be explicitly selected by the VPL programmer with the aid of the "|" operator. A goal that does not commit individually delegates its commit to its caller. Compensating actions are defined as clauses and can denote actions at local database systems.

Example 15 *(noncascading compensation (Example 9 in VPL))*

gta_{15} <– london and sailing.

london <– LTA_{flight} and LTA_{hotel} and LTA_{car} and LTA_{events} and
compensate(sell(trip_to_london)) | .

sailing <– LTA_{sea}.

Global transactions are not committed in these examples unless explicitly specifies by the placement of "|" at the conclusion of the gta. Thus, they can be involved in the composition of other global transactions.

4.1.4. Advanced Control Mechanisms

External constraints. External constraints can be easily represented in VPL. For example, the predicate "actual_time(Time)" returns in its argument the actual time. If a transaction t_i must be started only during office hours, we write:

... actual_time(T) & T greater 9:00 & T less 17:00 & LTA_i ...

The specification of constraints in VPL is thus straightforward.

4.2. General Transaction Control

We shall now provide examples of the use of the VPL communication mechanism. Using communication variables a wide range of control and data flow types can be programmed.

Let us define a predicate "call_LTA(LTA, Pre, Post, PID, State, C)" that calls a subtransaction and executes a pre- and a post-condition. "Pre," "Post," "PID," "State," and "C" are communication variables that can be shared between different activations of "call_LTA." "State" allows an external examination of the execution state of the local execution. It is set to 'SUCCEEDED' if the LTA has succeeded and the postcondition can be fulfilled. It is set to 'FAILED' either if the precondition cannot be fulfilled or if the LTA or the postcondition fails. The "test_pre" predicate guarantees that the precondition can succeed only once—preventing a second activation of the LTA, and on backtracking sets "State" to 'FAILED.' "C" signals to the independent process that it should commit. "PID" is automatically bound to 'SUCCEEDED' by VPL after the process has successfully completed or to 'FAILED' if the execution was unsuccessful.

We have modified the goal of the process such that it must wait until condition "C" allows it to commit.

We define the predicate "set" to write a value into a communication variable and to immediately commit this writing so it can be observed by other concurrent processes. The process must be started with type 'INDEP' because "call_LTA" can be used to start alternatives. Type 'DEP' would leave the caller dependent on all alternatives and would negate caller success if one alternative is rejected.

We implement "call_LTA" as follows:

```
call_LTA(LTA, Pre, Post, PID, State, C) <-
    process('LOCAL',
            test_pre(Pre) & LTA & Post & set(State,'SUCCEEDED') & C | ,
            PID,
            'INDEP').

set(CVar, Val) <- CVar =#= Val  | .

test_pre(Pre) :- Pre and compensate(set(State, 'FAILED')) | .
test_pre(Pre) :- set(State, 'FAILED') & fail.
```

4.2.1. Alternatives

k of n. Example 16 *(k of n (Example 2 specified in VPL))*

$gta_{16}(\ldots)$ <-
call_LTA(LTA_{opera}, true, true, PID_o, $State_o$, C_o = true) &&
call_LTA($LTA_{theatre}$, true, true, PID_t, $State_t$, C_t = true) &&
call_LTA($LTA_{concert}$, true, true, PID_c, $State_c$, C_c = true) &&
$control_{16}$($State_o$, $State_t$, $State_c$, C_o, C_t, C_c).

$control_{16}$('SUCCEEDED', 'SUCCEEDED', $State_c$, C_o, C_t, C_c) <-
C_c =#= false and C_o =#= true and C_t =#= true.

$control_{16}$('SUCCEEDED', $State_t$, 'SUCCEEDED', C_o, C_t, C_c) <-
C_t =#= false and C_o =#= true and C_c =#= true.

$control_{16}$($State_o$, 'SUCCEEDED', 'SUCCEEDED', C_o, C_t, C_c) <-
C_o =#= false and C_t =#= true and C_c =#= true.

The predicate "$control_{16}$" waits until it determines that a commit set has been reached, thus implementing implicit synchronization via goal-head unification. It then requests the C variable of the transaction extraneous to the selected commit set to be set to "false," thus triggering the abortion of the corresponding transaction. Finally, it requests the C variables of the selected transactions to be set to "true" at the end of the next transaction. This links the commitment of these transactions to the commitment of their governing transaction. On commitment of the governing transaction, all C variables are bound in an atomic step which activates all processes awaiting the setting of C to "true". Note that a goal "C_i = true" implements a synchronization point.

Selection of arbitrary patterns. Example 17 *(overlapping patterns)*

Let us extend Example 3 to include the reservation of a rental car. However, if a flight is booked, a bicycle may be reserved instead of a car. The global transaction is fulfilled if either t_{train}, t_{ship}, and t_{car}; t_{flight} and t_{car}; or t_{flight} and $t_{bicycle}$ can be booked.

$gta_{17}(\ldots)$ <-

call_LTA(LTA$_{ship}$, true, true, PID$_s$, State$_s$, C$_s$ = true) &&
call_LTA(LTA$_{train}$, true, true, PID$_t$, State$_t$, C$_t$ = true) &&
call_LTA(LTA$_{flight}$, true, true, PID$_f$, State$_f$, C$_f$ = true) &&
call_LTA(LTA$_{bicycle}$, true, true, PID$_b$, State$_b$, C$_b$ = true) &&
call_LTA(LTA$_{car}$, true, true, PID$_c$, State$_c$, C$_c$ = true) &&
control$_{17}$(State$_s$, State$_t$, State$_f$, State$_b$, State$_c$, C$_s$, C$_t$, C$_f$, C$_b$, C$_c$).

control$_{17}$('SUCCEEDED','SUCCEEDED',State$_f$,State$_b$,'SUCCEEDED',
C$_s$,C$_t$,C$_f$,C$_b$,C$_c$) <–
C$_f$ =#= false and C$_b$ =#= false and C$_s$ =#= true and
C$_t$ =#= true and C$_c$ =#= true.

control$_{17}$(State$_s$,State$_t$,'SUCCEEDED',State$_b$,'SUCCEEDED',
C$_s$,C$_t$,C$_f$,C$_b$,C$_c$) <–
C$_s$ =#= false and C$_t$ =#= false and C$_b$ =#= false and
C$_f$ =#= true and C$_c$ =#= true.

control$_{17}$(State$_s$,State$_t$,'SUCCEEDED','SUCCEEDED',State$_c$,
C$_s$,C$_t$,C$_f$,C$_b$,C$_c$) <–
C$_s$ =#= false and C$_t$ =#= false and C$_c$ =#= false and
C$_f$ =#= true and C$_b$ =#= true.

Without the use of communication variables, this transaction could not be formulated without violating the condition that a transaction can be started only once or without introducing artificial dependencies.

4.2.2. *Execution State Dependencies*

Success and failure dependencies. If the subtransactions of a Flex Transaction cannot be divided into groups based on success and failure dependencies, they must be tested via communication variables.

Example 18 *(IF-THEN-ELSE)*
The semantics of the following transaction are: if a flight can be booked to Paris (t$_{flight}$), then reserve a hotel (t$_{hotel}$) there; otherwise, book an overnight ferry to Budapest (t$_{ship}$). This can be specified in Flex via the dependencies t$_{flight}$ $\prec_S$ t$_{hotel}$ and t$_{flight}$ $\prec_F$ t$_{ship}$ and the set of commit sets {{t$_{flight}$, t$_{hotel}$}, {t$_{ship}$}}.

In VPL we represent this transaction as:

gta$_{18}$(...) <–
call_LTA(LTA$_{flight}$, true, true, PID$_f$, State$_f$, C$_f$=true) &&
call_LTA(LTA$_{hotel}$, State$_f$=true, true, PID$_h$, State$_h$, C$_h$=true) &&
call_LTA(LTA$_{ship}$, PID$_f$='FAILED', true, PID$_s$, State$_s$, C$_s$=true) &&
control$_{18}$(State$_f$, State$_h$, State$_s$, C$_f$, C$_h$, C$_s$).

control$_{18}$('SUCCEEDED', 'SUCCEEDED', State$_s$, C$_f$, C$_h$, C$_s$) <–

C_f =#= true and C_h =#= true and C_s =#= false.

$control_{18}$(State$_f$, State$_h$, 'SUCCEEDED', C_f, C_h, C_s) <–
C_f =#= false and C_h =#= false and C_s =#= true.

We shall now illustrate the specification of more complicated dependencies (as in ACTA) in VPL. Such dependencies can result in either optimistic or pessimistic concurrency control. When possible, we will opt for the pessimistic method, in which no transaction can be activated until it is guaranteed that the dependency can be satisfied. In Section 5.4, we will outline optimistic testing by means of event streams.

The implementation of these dependencies also depends on whether or not a local subtransaction commits individually and thus is of the compensatable type. For example, to guarantee commitment dependency (t_i $\mathcal{CD}$ t_j) for noncompensatable transactions, it is sufficient to test whether t_i has committed under the postcondition of t_j. However, if t_j is compensatable, the test must be performed under the precondition of t_j. Otherwise, the commitment of t_j cannot be prevented.

For the following examples, we assume noncompensatable subtransactions that are called with remote_call$_{NC}$ and delegate their commitment to the caller.

If the dependency is fulfilled, the gta succeeds; otherwise, the gta fails. Our implementation of the gta is reduced to show the dependency between t_i and t_j. Other transactions may also be called in the gta, possibly between t_i and t_j. We assume that communication variables are shared and thus can also be set elsewhere. Composition of several dependencies is possible by connecting their pre-, post- or commit conditions by logical AND.

Commit dependency. gta$_{CD}$ implements t_i $\mathcal{CD}$ t_j:

gta$_{CD}$(...) <–
call_LTA(LTA$_i$, true, true, PID$_i$, State$_i$, true) &&
call_LTA(LTA$_j$, true, F$_j$=true, PID$_j$, State$_j$, true) &&
control$_{CD}$(PID$_i$, F$_j$).

control$_{CD}$('SUCCEEDED', F$_j$) <– set(F$_j$, true).
control$_{CD}$('FAILED', F$_j$) <– set(F$_j$, true).

We have implemented $\mathcal{CD}$ such that t_j must wait until t_i has terminated before it can commit. Controlling the PID$_i$ differs from controlling State$_i$ in that PID$_i$='SUCCEEDED' implies that t_i was needed for the global transaction, whereas State$_i$='SUCCEEDED' indicates only that t_i succeeded (having reported prepared), but it is still unclear whether the global transaction will be committed or t_i will be compensated.

Strong-commit dependency. gta$_{SCD}$ implements t_i $\mathcal{SCD}$ t_j:

gta$_{SCD}$(...) <–
call_LTA(LTA$_i$, true, true, PID$_i$, State$_i$, true) &&

call_LTA(LTA$_j$, true, true, PID$_j$, State$_j$, true) &&
control$_{SCD}$(PID$_i$, PID$_j$).

control$_{SCD}$('SUCCEEDED', 'SUCCEEDED') <– true.
control$_{SCD}$('FAILED', PID$_j$) <– true.
control$_{SCD}$(PID$_i$, 'FAILED') <– PID$_i$='SUCCEEDED' &
signal(PID$_i$, 'ABORT').

We have implemented $\mathcal{SCD}$ by requiring t_j to succeed if t_i succeeds. If t_i fails, t_j may either succeed or fail. If t_i fails, t_j is compensated (its process is sent the 'ABORT' signal).

Abort dependency. gta$_{AD}$ implements t_i $\mathcal{AD}$ t_j:

gta$_{AD}$(...) <–
call_LTA(LTA$_i$, true, true, PID$_i$, State$_i$, true) &&
call_LTA(LTA$_j$, true, F$_j$=true, PID$_j$, State$_j$, true) &&
control$_{AD}$(PID$_i$, F$_j$).

control$_{AD}$('FAILED', F$_j$) <– set(F$_j$, false).
control$_{AD}$('SUCCEEDED', F$_j$) <– true.

t_j must not succeed if t_i aborts. If t_i succeeds, the execution state of t_j is not relevant.

Weak-abort dependency. gta$_{WD}$ implements t_i $\mathcal{WD}$ t_j:

gta$_{WD}$(...) <–
call_LTA(LTA$_i$, true, true, PID$_i$, State$_i$, true) &&
call_LTA(LTA$_j$, true, true, PID$_j$, State$_j$, C$_j$=true) &&
control$_{WD}$(PID$_i$, C$_j$).

control$_{WD}$('FAILED', C$_j$) <– set(C$_j$, false).
control$_{WD}$('SUCCEEDED', C$_j$) <– true.
control$_{WD}$(PID$_i$, C$_j$) <– cvar(PID$_i$).

t_j may commit only if t_i has not already failed. If t_i has already succeeded, t_j may commit. If t_i is still running, which is ensured by the "cvar" test, t_j may also commit.

Termination dependency. gta$_{TD}$ implements t_i $\mathcal{TD}$ t_j:

gta$_{TD}$(...) :–
call_LTA(LTA$_i$, true, true, PID$_i$, State$_i$, true) &&
call_LTA(LTA$_j$, true, F$_j$=true, PID$_j$, State$_j$, true) &&
control$_{TD}$(PID$_i$, F$_j$).

control$_{TD}$('SUCCEEDED', F$_j$) <– set(F$_j$, true).
control$_{TD}$('FAILED', F$_j$) <– set(F$_j$, true).

Exclusion dependency. gta$_{ED}$ implements t_i $\mathcal{ED}$ t_j:

$gta_{ED}(\ldots)$ <–
call_LTA(LTA$_i$, true, true, PID$_i$, State$_i$, true) &&
call_LTA(LTA$_j$, true, F$_j$=true, PID$_j$, State$_j$, true) &&
control$_{ED}$(PID$_i$, F$_j$).

control$_{ED}$('SUCCEEDED', F$_j$) <– set(F$_j$, false).
control$_{ED}$('FAILED', F$_j$) <– true.

Compare with the implementation of $_{AD}$.

Force-commit-on-abort dependency. gta_{CMD} implements t$_i$ $\mathcal{CMD}$ t$_j$:

$gta_{CMD}(\ldots)$ <–
call_LTA(LTA$_i$, true, true, PID$_i$, State$_i$, true) &&
call_LTA(LTA$_j$, true, true, PID$_j$, State$_j$, true) &&
control$_{CMD}$(PID$_i$, PID$_j$).

control$_{CMD}$('FAILED', 'SUCCEEDED') <– true.
control$_{CMD}$('SUCCEEDED', PID$_j$) <– true.

t$_j$ must succeed if t$_i$ fails. Otherwise, the execution state of t$_j$ is irrelevant.

Begin dependency. gta_{BD} implements t$_i$ $\mathcal{BD}$ t$_j$:

$gta_{BD}(\ldots)$ <–
call_LTA(LTA$_i$, set(F$_j$, started), true, PID$_i$, State$_i$, true) &&
call_LTA(LTA$_j$, F$_j$=started, true, PID$_j$, State$_j$, true).

The fact that t$_i$ has begun is signaled to t$_j$ by the fact that t$_i$ in its precondition sets F$_j$ to 'started.' This is tested in the precondition of t$_j$.

Serial dependency. gta_{SD} implements t$_i$ $\mathcal{SD}$ t$_j$:

$gta_{SD}(\ldots)$:–
call_LTA(LTA$_i$, true, true, PID$_i$, State$_i$, true) &&
control$_{SD}$(PID$_i$, F$_j$) &&
call_LTA(LTA$_j$, F$_j$=true, true, PID$_j$, State$_j$, true).

control$_{SD}$('SUCCEEDED', F$_j$) <– set(F$_j$, true).
control$_{SD}$('FAILED', F$_j$) <– set(F$_j$, true).

t$_j$ must wait until t$_i$ has terminated. This is tested in precondition of t$_j$.

Begin-on-commit dependency. gta_{BCD} implements t$_i$ $\mathcal{BCD}$ t$_j$:

gta_{BCD} <– LTA$_i$ & LTA$_j$.

Begin-on-abort dependency. gta_{BAD} implements t$_i$ $\mathcal{BAD}$ t$_j$:

gta_{BAD} :– LTA_i.
gta_{BAD} :– LTA_j.

Weak-begin-on-commit dependency. If implemented pessimistically, gta_{WCD} equals gta_{SD}.

4.2.3. Data Dependencies

Communication variables can serve to model data dependencies. For example, output data can be shared as communication variables between several processes which monitor their states and values as the basis of decisions. The output variable of the remote_call is a communication variable, the value of which can be tested by the VPL program. For example, this value can be used to constrain the total price of reserved travel. Since the representation of queries was not discussed in detail in this paper (the remote_call was assumed), we will not provide any examples of the process.

4.2.4. User Interaction

Unlike Prolog, VPL is capable of effectively interpreting input and output. User input is represented as a stream of communication variables. When a value is committed in the stream, it becomes visible and persistent.

Let us assume that a transaction requires an interactive retrial:

```
retry(User) :- #U1 & User=#=['retry ?'|U1]  | U1 = [yes|U2].
```

Depending on the user's input, the predicate "retry" succeeds or fails. It can be used to trigger the retrial of a transaction.

4.2.5. Advanced Control Mechanisms

Timeout. All parallel operators of VPL guarantee a fair execution and thus can be used to specify a timeout ("sleep(N)" waits for N seconds). The following example illustrates the "call_with_timeout" procedure that calls a transaction T. If the timeout expires before T succeeds, "call_with_timeout" fails and causes the abort of T. Ctrl is a communication variable and is shared between the two OR parallel branches.

```
call_with_timeout(T, N, Ctrl) ::– T & Ctrl=#=true | .
call_with_timeout(T, N, Ctrl) ::– sleep(N) & Ctrl=#=false | fail.
```

Retry condition. An example of this retry condition has been provided in Section 4.2.4.

To control the retrial of a transaction, we write:

... test & LTA_i ...

test :– true.
test :– retry_condition.

The first test clause will succeed as required for the first (forwards) execution of t_i. If the execution of t_i fails, backtracking occurs and "test" is asked for another solution. The second test clause is then tried; this calls the retry condition. If retrial succeeds, forwards execution is started and t_i is re-entered. The number of retries of t_i is equal to the number of solutions.

For example, a retry condition that is fulfilled N times can be specified as:

$retry_N$(0) :– | fail.
$retry_N$(N) :– true.
$retry_N$(N) :– N1 is N-1 & $retry_N$(N1).

Repetition. If a subtransaction must be reused several times within a transaction, we can simply rewrite the subtransaction the required number of times.

Dynamic interrupts. VPL's primitive "signal" can be used to dynamically interrupt running processes.

Current state. As the values of communication variables can be observed by parallel processes, they can readily be used to check the results or states of transactions performed to date.

Dynamic changes. Since Prolog-based languages permit dynamic program changes, such changes can actually be programmed. Advanced programming techniques in Prolog are discussed in [27].

5. Advanced Aspects of VPL-Based Transaction Specification

An MDBS transaction consists of more than simply the specification of the control and data flow between parallel activities.

5.1. Fault-Tolerance

Using the VPL recovery mechanism, a gta can be started as an 'INDEP' process, guaranteeing that the gta will eventually be performed. The execution of distributed transactions is thus rendered independent of system and site failures. Perpetual activities can thus be modeled as 'INDEP' processes that never terminate. Furthermore, I/O via communication variables is highly reliable, and user input is never lost.

5.2. *Delayed Compensation*

A compensating action can be specified as an unbound communication variable. This communication variable can later be bound, permitting the delayed specification of a compensating action in a manner analagous to the formation of a prepared state.

5.3. *Advantages of Prolog*

A number of advantages of logic-based languages have been discussed throughout this paper. Additonal advantages include:

- VPL can be used to specify control and data flow, as it is a query and transaction control specification language.
- A VPL specification is runnable.
- The formulation of knowledge can be accomplished using the rules of Prolog.
- Scheduling can be improved using metaschedulers which incorporate information regarding the duration of transactions [19] or by allowing the VPL system to exploit the maximum parallelism allowed by a specification.
- Many formulations are possible for any query.
- Backtracking within VPL involves the automatic trial of alternative solution paths. The "IF t_{flight} THEN t_{hotel} ELSE t_{ship}" case originally presented in Example Example 18 must be representative of this capability, which will not be further addressed in this paper due to length limitations.

Example 19 *(IF-THEN-ELSE with Backtracking)*
In Example 18, after the success of t_{flight} and the failure of t_{hotel}, the commit set $\{t_{ship}\}$ cannot be reached because of the failure dependency $t_{flight} \prec_F t_{ship}$. This dependency states that t_{ship} must not be tried unless t_{flight} has failed. If we allow a slightly different interpretation of this dependency, we may model the same example in a declarative way, allowing backtracking:

gta_{19} :- LTA_{flight} & LTA_{hotel}.
gta_{19} :- LTA_{ship}.

The failure of t_{hotel} causes backtracking and thus the abortion of t_{flight}, which in turn either causes its abort at the local database system or its compensation. In either case, we may state that t_{flight} is unsuccessful and allow the solution $\{t_{ship}\}$.

If we extend Example 18 by the clause

$control_{18}$('SUCCEEDED', 'FAILED', $State_s$, C_f, C_h, C_s) <-
    set(C_f, false).

the same behavior can be achieved. Setting C_f to 'false' triggers the abortion of the process, calling LTA_{flight} and thus setting PID_f to 'FAILED.' Thus, LTA_{ship} can be started.

5.4. *Reliable Event Streams*

A stream of communication variables can be shared among several active local transactions, and each transaction can be asked to write its significant events on that stream. This stream can then be considered as a *history* and used for optimistic concurrency control. The appropriate modification of the call_LTA predicate is readily accomplished through the addition of an event stream argument. Every state transition of the local transaction is then written into this stream.

6. Conclusions

The processing of MDBS transactions imposes a number of new requirements on traditional transaction models. Moreover, an advanced language is needed for the specification of MDBS transactions.

In this paper, we have shown how a general-purpose, logic-based coordination language can serve this purpose. In VPL all control flow patterns can be specified, often in multiple ways. All specifications using VPL can be executed directly. Using VPL, a user is thus no longer dependent on predefined functions and is capable of formulating any MDBS transaction.

Prototype implementations of our coordination tools are available for UNIX workstations connected by local or wide area networks, supporting the IP protocol (send e-mail to eva@mips.complang.tuwien.ac.at).

Our next step will involve the design and implementation of a graphical user interface that will expedite VPL programming. We also plan to develop protocols tailored for the communication of very large amounts of data with the intention of integrating multimedia data.

Acknowledgments

We acknowledge the support and encouragement of Manfred Brockhaus, the chair of the Computer Languages Department at the TU Wien, and the helpful comments of Alexander Forst, Wei Liu, Herbert Pohlai, Konrad Schwarz, and Thomas Tschernko on this text.

Notes

1. Due to space limitations, object-orientation is not discussed in this article. Details may be found in [18], [5]
2. For implementation of the fault-tolerant communication protocol, see [17].

References

1. P. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
2. K. P. Birman, "The process group approach to reliable distributed computing." *Communications of the ACM*, 36(12), pp. 37–53, 1993.
3. Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz, and A. Silberschatz, "On rigorous transaction scheduling." *IEEE Transactions on Software Engineering*, 17(9), pp. 954–960, 1991.
4. A. Buchmann, M. T. Öszu, M. Hornick, D. Georgakopoulos, and F. A. Manola, A transaction model for active distributed object systems. In A. K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 5, pp. 123–158. Morgan Kaufmann Publishers, 1992.
5. O. Bukhres, A. Elmagarmid, and e. Kühn, Advanced languages for multidatabase systems. In O. Bukhres and A. K. Elmagarmid, editors, *Object-Oriented Multidatabase Systems*. Prentice-Hall, 1995. to appear.
6. O. Bukhres, e. Kühn, and F. Puntigam, A language multidatabase system communication protocol. In *Proceedings of the 9th International Conference on Data Engineering*, pp. 633–640. IEEE Computer Society, April 1993.
7. P. Chrysanthis and K. Ramamritham, ACTA: The SAGA continues. In A. K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 10, pp. 351–397. Morgan Kaufmann Publishers, 1992.
8. E. Dijkstra, "Guarded commands, nondeterminacy, and formal derivation of programs." *Communications of the ACM*, 18(8), pp. 453–457, 1975.
9. W. Du, A. Elmagarmid, and W. Kim, "Maintaining quasi serializability in multidatabase systems." In *Proceedings of the 7th International Conference on Data Engineering*, pp. 360–367, Kobe, Japan, April 1991.
10. A. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz, "A multidatabase transaction model for InterBase." In *Proceedings of the 16th International Conference on Very Large Data Bases*, pp. 507–518, Brisbane, Australia, August 1990.
11. A. K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, 1992.
12. A. Forst, e. Kuhn, and O. Bukhres, "General purpose work flow languages." *International Journal on Parallel and Distributed Databases*, 3(2), pp. 187–218, 1995. Special Issue on Software Support for Work Flow Management, Kluwer Adademic Publishers.
13. A. Forst, e. Kühn, H. Pohlai, and K. Schwarz, "Logic based and imperative coordination languages." In *Seventh International Conference on Parallel and Distributed Computing Systems, PDCS'94*, pp. 152–159, Las Vegas, Nevada, October 6-8 1994. ISCA, ACM, IEEE.
14. H. Garcia-Molina and K. Salem, "Sagas." In *Proceedings of the ACM SIGMOD Annual Conference*, pp. 249–259, San Francisco, May 1987.
15. D. Georgakopoulos, M. Rusinkiewicz, and A. Sheth, "On serializability of multidatabase transactions through forced local conflicts." In *Proceedings of the 7th International Conference on Data Engineering*, pp. 314–323, Kobe, Japan, April 1991.
16. J. Gray, "The transaction concept: Virtues and limitations." In *Proceedings of the VLDB (Very Large Databases)*, pp. 144–154, Cannes, France, September 1981.
17. e. Kühn, "Fault-tolerance for communicating multidatabase transactions." In *Proceedings of the 27th Hawaii International Conference on System Sciences (HICSS)*, pp. 323–332, Wailea, Maui, Hawaii, January 4–7 1994. ACM, IEEE.
18. e. Kühn, "A universal model for the coordination of distributed systems." Technical report, University of Technology Vienna, 1994.
19. e. Kühn, W. Liu, and H. Pohlai, "Scheduling transactions on distributed systems with the VP_L engine." In *In: Proceedings of the Second Biennial European Joint Conference on Engineering Systems Design, ESDA'94*, pp. 335–347, London, England, July 4–7 1994. The American Society of Mechanical Engineers (ASME).
20. e. Kühn and T. Ludwig, VIP-MDBS: A logic multidatabase system. In *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems*, pp. 190–201, Austin, Texas, December 1988. IEEE Computer Society Press. Also included as part of the text of a new IEEE Computer Society Press tutorial *Multidatabase Systems: An Advanced Solution for Global Information Sharing*, by A. R. Hurson, M. W. Bright and S. Pakzad, 1993.

21. e. Kühn, H. Pohlai, and F. Puntigam, Concurrency and backtracking in $V^{vienna}_{Parallel\ Logic}$. *Computer Languages*, 19(3), pp. 185–203, July 1993.
22. e. Kühn, H. Pohlai, and F. Puntigam, "Communication and transactions in $V^{vienna}_{Parallel\ Logic}$." *Computers and Artificial Intelligence*, 13(4), pp. 301–319, 1994.
23. e. Kühn, F. Puntigam, and A. K. Elmagarmid, "An execution model for distributed database transactions and its implementation in VPL." In *Proceedings of the International Conference on Extending Database Technology, EDBT'92*, pp. 483–498, Vienna, March 1992. Springer Verlag, LNCS.
24. Y. Leu, *Flexible Transaction Management in the InterBase Project.* PhD thesis, Purdue University, August 1991.
25. J. E. Moss, "Nested transactions: An introduction." In B. Bhargava, editor, *Concurrency Control and Reliability in Distributed Systems*, pp. 395–425. Van Nostrand Reinhold, 1987.
26. E. Shapiro, "The family of concurrent logic programming languages." *ACM Computing Surveys*, 21(3), pp. 413–510, September 1989.
27. Ehud Shapiro, *The Art of Prolog*. The MIT Press, 1986.
28. L. Suardi, M. Rusinkiewicz, and W. Litwin, "Execution of extended multidatabase SQL." In *Proceedings of the 9th International Conference on Data Engineering*, pp. 641–650, Vienna, Austria, April 1993. IEEE Computer Society.
29. H. Wächter and A. Reuter, The ConTract model." In A. K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 7, pp. 219–263. Morgan Kaufmann Publishers, 1992.