BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-93-M10

"Specification Environment For Multidatabase Applications"

by

Noela V. Nakos

SPECIFICATION ENVIRONMENT FOR MULTIDATABASE APPLICATIONS

Noela V. Nakos
Department of Computer Science
Brown University

Submitted in partial fulfillment of the requirements for the Degree of
Master of Science in the Department of Computer Science at Brown University

May 1993

This research project by Noela V. Nakos is accepted in its present form by the Department of Computer Science at Brown University in partial fulfillment of the requirements for the Degree of Master of Science.

Stanley B. Zdonik

Professor Stanley B. Zdonik
Advisor

5/19/93

Date

SPECIFICATION ENVIRONMENT FOR MULTIDATABASE APPLICATIONS

Noela V. Nakos
Department of Computer Science
Brown University

May 4, 1993

ABSTRACT

A specification environment for the development of Multidatabase applications is discussed. The architecture of the environment, its features and the communication scheme for the Multidatabase system are reviewed along with a detailed description of a language for the specification of planning applications.

CHAPTER 1 : INTRODUCTION

1.1 Origin of Multidatabases

The need for accessing large amounts of data not always stored in a single data repository is a common need application developers face today. A typical example is that of a Travel Agency in which applications access data from different enterprises, airlines, hotels, car rentals. The data is thus stored in different databases but from the travel agent's perspective it must be accessed as if stored in a single repository of information. The main difficulty the application developer is confronted with is the existence of a multitude of databases each of which may support its own concurrency control protocol, data model and data manipulation language.

A multidatabase is a collection of heterogeneous, independent local databases containing information that applications will query or update together. The purpose of a multidatabase is to allow access to data which is stored in disparate databases in a seemingly transparent manner. That is, data across database boundaries is perceived as belonging to only one database. Multidatabases are particularly useful in cases where it is not practical or even possible to transfer the data from all local databases into a single repository of information. Example: the data belongs to different enterprises.

1.2 The Multidatabase Environment

Figure 1.1 shows the multidatabase environment. In this environment a multidatabase application is responsible for accomplishing a series of tasks while maintaining data consistency. Futhermore, we assume the existence of independent applications which do not access the multidatabase interface but interact directly with a particular local database. In the example of the travel agent, we assume that an airline does not have to be aware of the existence of the multidatabase and can access the local repository of information independently.

Environment assumptions :

a. Heterogeneous databases - The different local databases may support different concurrency control protocols, data models and data manipulation languages.
b. There is a uniform interface to the information which is accessed through the multidatabase.
c. No design modifications should be necessary on the local databases in order to support the multidatabase.
d. The local databases may be distributed in any way.
e. Multidatabase transactions are not restricted to executing only certain transaction types, e.g. read-only access.

1.3 Planning Applications

Some of the issues that multidatabases must resolve are directly related to the needs of the kinds of applications that make use of them. Planning applications encompasses most of these difficulties. In a planning application we encounter a series of tasks which

affect different objects in different local databases. These tasks may interact with each other as they access the same data. Moreover, independent transactions and other multidatabase applications may access the same data which could interfere with the progress of some task in the context of the planning application. Furthermore, planning applications are usually long-lived which implies that the effects accomplished by a task in the planning application might become invalid by the time the application reaches completion. Example: A travel agent preparing a trip for a customer makes use of the multidatabase as he/she reserves flights, hotels and car rentals in the context of one application. Since, a trip might take from hours to months the application is active throughout this time, the multidatabase must guard against changes that can make one or more of the accomplished application tasks invalid and must react accordingly.

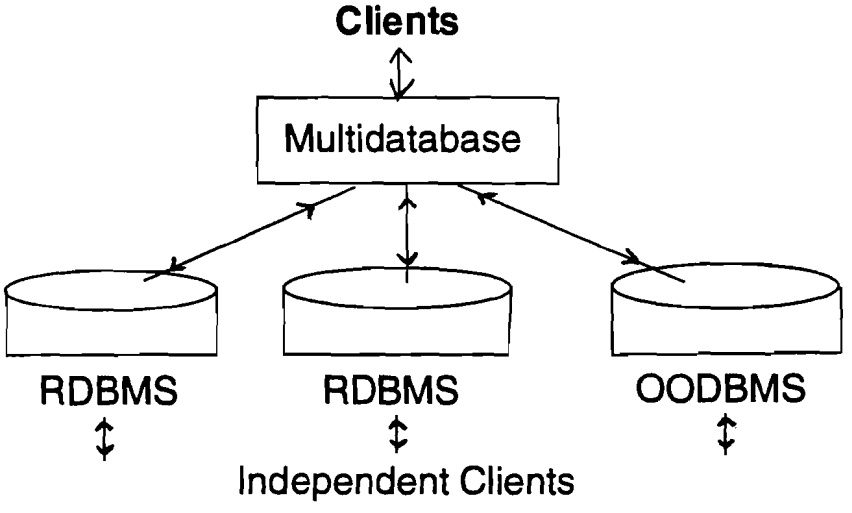

Figure 1.1 : Multidatabase Environment

1.4 Defining a Multidatabase

In the work of Marian Nodine [MNF92] a multidatabase is defined as consisting of two levels. These two levels are the local and the global level. Based on her definition, we call the local level the level that consists of the series of local databases which are being integrated through the Multidatabase. It is at the local level where all the information accessed by the multidatabase is stored and it is the local level the component responsible for data persitence.

Meanwhile, the global level is the component in charge of allowing users to access the data across the local level in a transparent manner, i.e. as if accessing data from a single repository of information. Thus the global level is responsible for presenting the user with a simple and structured way of accessing the data stored in the local databases. Furthermore, given the heterogeneity assumption (i.e. the local repositories of information are heterogeneous) the global level must also ensure that the information in the multidatabase is consistent. Following the traditional approach for databases, consistency is maintained by following the ACID properties. However, multidatabases are of particular use in the context of long-term transactions in which maintaining atomicity is no longer feasible since locking resources for the lifetime of an application implies preventing other applications from accessing data for very long periods of time. Thus the global level must enforce a mechanism for maintaining consistency while not relying on atomicity.

In the remainder of this paper I will concentrate on the global level of an implementation of a Multidatabase under a project called **Mongrel** directed by Marian Nodine and Stan Zdonik at Brown University in Providence, R.I.. In particular I will describe the user Interface and the language for defining tasks in a multidatabase. Note that I will reference different components of the multidatabase which are based on the above-described two-level definition. However, I will not go into detail defining them, further information can be found in Marian Nodine's papers listed in the references.

CHAPTER 2 : DEFINITIONS AND REQUIRMENTS

2.1 Interactions

Our first step is to understand what a task is in the context of multidatabase applications. A task in the context of multidatabases is a goal for whose accomplishment we must access one or more repositories of information. A task then, can be divided into a series of subtasks or subgoals each of which accesses data on a different local database. Example: Consider the case of the travel agent preparing a trip for a customer who wishes to fly from Providence to San Francisco on the 1st of March returning to Providence on the 15th of the same month. The customer also wishes to have a hotel reservation and a rental car. Thus the travel agent has a well-defined task which places a series of constraints and dependencies on the actions that can be taken in the process of fulfilling the task, e.g. a hotel reservation is useful only if it is done for the customer's length of stay in San Francisco, the same is the case with the car rental. We can then divide our task into a series of subtasks. These subtasks include a flight reservation, a hotel reservation and a car rental reservation. We notice however, that the subtasks are not independent, on the contrary the series of constraints and dependencies placed on the task must be observed by the subtasks as they accomplish their goals.

In the context of Marian Nodine's work a task is represented by what we call an *Inter-*

action. As it is the case with tasks, an Interaction can be further divided into subtasks which in our work are represented by *Global Transactions (GT)*. A GT then contains the definition of the actions that must take place for the Interaction to accomplish a subtask. In other words, an Interaction is a program which defines the actions that must take place in order to accomplish a task. Furthermore, an Interaction is responsible for maitaining an enforcing the task's constraints and it does so by placing constraints on the data manipulated by the GTs and by defining the order in which GTs should be executed. An Interaction keeps the information relative to its state in the accomplishment of the task in a series of variables which are shared by the GTs.

Structure

Graphically we can describe an Interaction as in Figure 2.1 The figure depicts the time-dependency/execution-dependency among the different subtasks or GTs that make up the Interaction. These dependencies are specified by the task definer. In the figure we see GT1 executing before GT2, GT3 and GT4. We further see that GT3 and GT4 execute in parallel. Thus, we gather that there must exist some kind of dependency which forces GT1 to commit before GT2 can begin, and GT2 to commit before either GT3 or GT4 can begin. However, the fact that GT3 and GT4 execute in parallel implies that there exists no dependencies among the subtasks they try to accomplish. Yet another sort of dependency which is not established by the nature of the task is depicted in the figure. In it we see the GTs accessing the variables which store the state of the Interaction. Thus, the GTs are sharing data by reading or writing to the state of the Interaction. This kind of dependency comes from the order of access, e.g. if GT4 reads from a variable which was written by GT3 then GT4 depends on the execution and validity of GT3. We have therefore, establish two kinds of dependencies which I will call *explicit* and *implicit* dependencies. An explicit dependency is one that comes from the specification of the task the Interaction tries to accomplish, e.g. we can not reserve a hotel unless we have a plane ticket. An implicit dependency is one

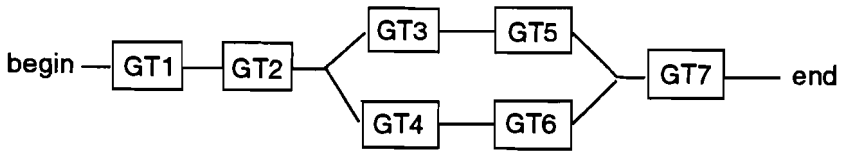

Figure 2.1 : Sample InterAction Execution

that arises from the way in which the task definer accesses the state of the Interaction through the GTs.

In our work GTs are further subdivided into what we call *global subtransactions* (GST). A GST is defined as the portion of a GT which acts on one and only one local database. We also assume that a GT cannot have more than one GST acting on a particular local database. Finally, we assume that GSTs begin executing after their parent GT starts execution and commit atomically when their parent GT commits. The structure and execution of a GT is depicted in Figure 2.2.

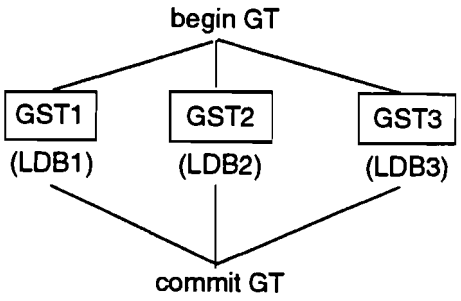

Figure 2.2 : Sample Global Transaction Structure

A GST (global subtransaction) which sometimes I will refer to as an action is defined as an ordered set of *Steps*, which are procedure calls on the local databases.

Actions and Steps

The following definitions are based on Marian Nodine's work. Extended explanations of what follows can be found in her papers in the references. An action is a partially ordered set of steps. Each of the steps accomplishes the same objective, thus exactly one step must succeed for the action to succeed. Example: Referring back to our travel agent and the customer planning a trip to San Francisco, let's assume that the customer is a frequent flyer member of Northwest airlines and therefore Northwest is his/her first choice, however, if there are no seats/flights on Northwest for those days, he/she is willing to fly Delta, United, USAir in that order. Thus the action consists of reserving the plane ticket, furthermore, the action has an order in terms of with whom the reservation should take place, therefore Northwest should be tried first, however it is enough for one reservation to go through for the action to succeed. Note that the action here is vital, that is the customer can not go to San Francisco unless he takes a plane. On the other hand there are actions of non-vital nature, e.g. may be a car rental is not necessary, then the action which represents the car reservation might have as an option the NULL step in which case the action succeeds even if there are no cars available for rental.

We define a step as a cohesive unit of work which performs a series of queries and/or updates on a single database. Each step in our implementation is a procedure call on the Local Database's Step Library. A Step Library is a repository of steps, i.e. each step that the multidatabase can execute on a local database is defined in the Step Library. Thus, a flight reservation is considered a step and the particulars on how the step is implemented, i.e. which transaction gets executed in the local database are hidden from the task definer through the Step Library.

Data Consistency

A global transaction in this work commits atomically once all of its steps have executed, i.e. once all of its actions have succeded. This however, implies that the effects that a global transaction had on data across the multidatabase becomes visible to other GTs. These other transactions can be independent transactions or part of the same or other Interactions before the Interaction that committed the GT commits itself. Data consistency then becomes an issue to resolve. On one hand if the Interaction which owns the committed global transaction aborts we must provide a means for rolling back the effects of the committed global transactions that it owns; on the other hand other global transactions which belong to other Interactions or that are independent will have read data that might no longer be valid. As for the former issue we resolve it by semantically undoing committed GTS. That is, by making use of compensating transactions [MGS]. The second issue is a bit more complicated, if the transaction which has read data that has now become invalid is an independent transaction we have no means of rolling it back to a consistent state; however, for the case of GTs that have read the invalid data and that belong to other Interactions, there are ways of avoiding inconsistency. However the solution is not provided automatically by the system, the task definer must provide for reacting to inconsistencies through the setting of what we call *weak conflicts*.

Conflicts

There are two kinds of conflicts that we are faced with and which we denominate as *strong conflicts* and *weak conflicts*. Strong conflicts are those that are set in order to preserve GT isolation, i.e. "Strong conflicts define what sets of actions must be done atomically in the multidatabase, and consequently must be executed as a single atomic global transaction" [MND92]. Weak conflicts specify conditions on some piece of data in a local database which must be preserved for a span of time in the Interaction's

lifetime so as to maintain the Interaction's validity. Furthermore, a weak conflict must also establish a set of measures to be taken in the event of a set/subset of those conditions being violated. Thus, weak conflicts do not prevent the conditions from being violated but instead establish measures to be taken in case certain conditions which invalidate all or part of an Interaction occur. Example: Consider the example of the travel agent. Suppose that a customer got a plane reservation with United Airlines for the 1st of March on flight 184. A weak conflict could then be set specifying that in the case of the flight being cancelled before the departure date, the Interaction be notified and the GT that made the flight reservation be rolled back and retried from the beginning. Weak conflicts are modelled following the ECA structure [event-condition-action].

Events

An event can be described as a guard that places a condition on some piece of data. In the case, of the condition being set or violated the event is raised. Events are used for two different purposes. On one hand an event being raised can be used as a triggering mechanism to take action, thus it is of use in flow control. For example: consider an event specified on flight 184 which establishes that once the flight departed the hotel reservation must be confirmed. Thus, in this case the event triggers the execution of some piece of the Interaction. There is no violation endangering the validity of the Interaction. On the other hand an event being raised can be used to determine a violation that invalidates part or all of the Interaction. For example: an event could specify that in the case of flight 184 being cancelled the GT which originally made the reservation must be rolled back and a new reservation should be made.

2.2 Requirements on Interactions

From our previous discussion we can set forth a series of requirements for our Interactions in the context of multidatabases.

A. Semantic Atomicity: Given the nature of multidatabase applications, in particular their long-term nature, we can not follow the traditional atomicity requirement placed on traditional short-term transactions. "Because of the all-or-nothing properties traditionally associated with transactions, multidatabase tasks are expected to executed in a way that perserves 'semantic atomicity'. Semantic atomicity states that a transaction either runs or is left in a state that is semantically equivalent to some state that would have been reached if the original transaction had not run at all." [MND92]

B. Procedural: In traditional transactions, information and control flows within a single, isolated transaction definition. In the multidatabase environment the task definer must have ways of specifying not only how information and control flows within a single GT but also how it flows among the different GTs.

C. Flexibility: The Interaction definer should be able not only to specify what should be done but which are the preferences and how to go about trying them out so as to accomplish the most acceptable solution.

D. Reactive: An Interaction must be able to react to the changes in the local databases. Multidatabase applications are more vulnerable to the evolving data than traditional transactions, mainly due to their long-term nature. Thus, we must provide for the Interactions to be aware of those changes as they might affect the validity of part or all of the Interactions. Furthermore, in the event of changes invalidating part or all of an Interaction, the system must provide for means of taking the Interaction back into a valid and consistent state.

E. Interactive: Multidatabase applications are characterized by accessing disparate

data over long periods of time. The applications thus, tend to evolve over time as data in the local repositories change. A direct consequence of this fact is that it is difficult to predict the path that an Interaction will take over time. Thus the task definer is confronted with the need to interact and code as the Interaction progresses.

F. Persistent: The information needed for the execution of an Interaction can not always be stored in volatile memory. The main two reasons for this constraint are: first multidatabase applications tend to be long-term, and keeping information in volatile memory can cause loss of information, second due to the nature of multidatabase applications it is not always true that the Interaction will be active throughout its lifetime. Thus the data that specifies the state of an Interaction as well as information relative to the Interaction's validity must be stored in persistent storage so as to be accesible at all times either when the Interaction is active or inactive. Example: In the event of Flight 184 being cancelled the Interaction must find out about it whether it is active or inactive.

G. Compensability: In the context of multidatabase applications we can no longer sustain atomicity as a necessary condition for the execution of the entire Interaction. However we do expect the all or nothing effect to be maitained in the context of Interactions. That is, we expect the Interaction to either reach completion or to have no effect on the data it accessed. In our work we allow GTs to commit prior to the commit of the entire Interaction, then, if the Interaction later aborts we must provide a way of undoing the effects of all committed GTs that belong to that aborted Interaction. In order to restore the information accessed by committed GTs to a consistent state we use compensation. Thus in our work for every GT there exists a *compensating* GT, e.g. for a plane reservation there is cancellation of the reservation. A drawback to compensation is that we assume that it is always possible to take the compensating transaction to completion which might not always be true. Consider the case in which the travel agent made a reservation on an airline company and paid for the ticket, and

further suppose that the airline company later goes bankrupt, the reservation might be undoable, however it might not always be possible to get the money back.

H. Partially Undoable: There are cases where a condition is violated, which makes an already committed global transaction invalid. However, this might not affect the execution of the whole Interaction, thus we want to have a way of rolling back only the invalidated pieces of work and not the whole Interaction.

I. Identity: Interactions have a unique identifier that lasts for their lifetime. This identifier is a requirement since all communication within the multidatabase system will make use of it for identification purposes.

J. Access to Local Databases: There must be provisions to allow access to local databases through the Interaction specification in a non-restrictive manner, e.g. read-only access is not acceptable.

K. Isolation: Each task should be able to specify and enforce its own isolation requirements.

CHAPTER 3 : TaSL – DEFINING AND MONITORING TASKS

Having reviewed some of the issues and terminology and being aware of the requirements placed on Interactions we will move on to describing how TaSL (Task Specification Language) provides for the specification of Interactions as well as how it allows the task definer to monitor the tasks. We will review how the requirements are met and what exactly is the task definer confronted with. Please note that TaSL is not an end-user environment.

TaSL has been designed and implemented following a client/server architecture. The server acts as a permanent liason between the applications and the rest of the multidatabase system. All data relative to the execution and state of applications is stored and access via remote procedure calls (rpcs) which are dispatched by the server. Given the kinds of long term applications supported in the multidatabase, to ensure data and application persistence we make use of a local database to keep all system-wide and application relative data. The client on the other hand provides the interface to the system, the language and its interpreter. In what follows we will review the design and implementation of the TaSL architecture. Note: The design here presented has been partially implemented.

3.1 The Server

The server starts at system start up and remains active throughout the lifetime of the system. System in this context refers to our implementation of the multidatabase Mongrel. Applications access and update their data and the system data via the Server. Communication between the rest of the system and the existing applications is accomplished also through the server. Thus, whether an application is active or inactive the system is able to communicate with it via the Server into its data. Thus, if an application becomes inactive, all actions affecting its execution, such as the raising of an event that was set at some earlier point by the application, will be known to the Interaction whenever it becomes active again, allowing it to take immediate action. To accomplish this communication, the server maintains data relative to the system and to the applications together with a series of procedures to access the data through remote procedure calls.

The Server, thus, performs three basic functions:

- Access of the application to the data.
- Communication of the rest of the system with the applications.
- Concurrency Control.

System and Applications data

Data persistently stored is divided into System and Applications data.

System Data

In a multidatabase many changes can happen at the local level which affect the execution and definition of new tasks. For example new databases are added, old ones are dropped, new procedures for accessing the local databases (Steps) are created, some

can be edited or even dropped. Thus, the global level must assist the user in being aware of the available resources. It is also the global level's responsability to keep the applications from trying to access non-existent resources. In order to fulfill these requirements TaSL maintains a repository of information on the available databases, their steps and arguments to the steps. Moreover, as we will see in section 3.2 TaSL also allows the user to access this information through a user-friendly graphical interface.

Keeping this sort of system data does not always, however, prevent the user from trying to access non-existent resources. There exists sometimes a small window of time where data at the local level changes, a database is dropped for example and by the time that information reaches the global level actions could have been sent off for execution through the system. There are ways to prevent this from happening by syncronizing the local and global level, however, that would mean disallowing one of our basic requirements that establishes the local databases independence.

System data is divided into two classes, a listing of the available databases and a listing of the available procedures on them. In our implementation we used ObjectStore [OS] as our local repository management system. ObjectStore [OS] is an object-oriented database management system. In it we defined the two classes in a parent-child relationship. Each instance of the database class owns one or more instances of the Step class. This configuration was originated from the fact that in our system a step belongs to one and only one local database and a local database has one or more steps. At the same time for each step a list of the arguments it takes together with their types is maintained and used by the TaSL language interpreter which is part of the client component, for type-checking. The interpreter accesses this data through a well-defined set of methods which belong to the classes.

As previously mentioned, this system data is subject to change through its lifetime. We designed then a mechanism for maitaining this data updated. To do so, we created a series of methods acting on the classes for creation and update of the data. This

methods are accessed through remote procedure calls from the Step Libraries located at each local database. A change to a step, the creation of new ones, the disappearance of a local database triggers the communication between the Step Library and our local repository which keeps our data updated on those changes.

Application data

Each application defined and executed in the multidatabse will create and access information relative to its state, to events being raised, etc. throughout its lifetime. Such information thus, must persist. Therefore, TaSL keeps yet another repository of information for this kind of data. This information is created and owned by each of the existing applications.

The application data is kept in seven different classes each of which provides a series of methods for accessing and updating this data.

(A) The Interaction class. A requirement in our system is to provide Interactions with unique identities. These unique identifiers are known throughout our implementation of the system and are used for communication among the different system components. TaSL, thus, keeps this identity stored together with the rest of the Interaction's data in the Interaction's class. This identifier is given to the Interaction by the Interaction Manager (IM) portion of the Concurrency Control Manager (CCM, global component). The identifier is created in response to TaSL's request to the IM to begin an Interaction. For information on the CCM component please refer to [MND92].

Other information on Interactions such as their name, the code defined in them, a listing of the global transactions executed, etc is also stored in the Interaction class. As it was the case with the system data, there exists in the Application data parent-child relationships. The Interaction instances own the data stored in instances of the other application data classes. Moreover, the Interaction is our entry point into the

application data repository.

(B) State Variables class. In the process of defining an Interaction the task definer creates and updates variables that will hold the state of the Interaction. These variables are thus, used throughout the lifetime of the Interaction and are therefore stored persistently. When a variable is created by the task definer the TaSL interpreter stores the name of the variable, its type and initial value if any in the application database by making a remote procedure call to the appropriate method of the state variable class. Each Interaction maintains a list of state variable instances that belong to its definition and execution.

Once a state variable is created, the task definer can use it throughout the life of the application and since it is stored persistently it will be available whether the application has been active since its creation or not. When a variable is updated in the execution of a global transaction, the TaSL interpreter communicates via remote procedure calls with the methods provided for that purpose in the state variable. These methods include type-checking and updates to the readers/writers list of the state variable class.

The readers/writers list of the variable class is created at the time the variable is created and since then it is updated to keep track of the global transactions that read or write from/to the variables. Such a list is maintained in order to reconstruct information on implicit dependencies. Example: Given a GT which reads a variable that another GT wrote a dependency is created among these two transactions. This sort of dependency is used along with explicit dependencies in reconstructing the execution tree in order to assist the Recovery Manager portion of the system in determining the actions that must be taken if rollback becomes necessary for returning an Interaction to a consistent state.

(C) Global Transaction Class. Each Interaction instance owns a list of GT instances. That is, all GTs executed during the lifetime of the application are stored permanently

in the Global Transaction class. The Interaction has a list of them, this list is an ordered list following execution order.

Each global transaction created or otherwise executed in the flow of execution of an Interaction is stored in the Global Transaction Class and added to the list of GTs owned by the Interaction. The most important pieces of data stored for a GT are: its id which is created at the time the GT starts execution and it is known throughout the system; a local identifier which is used only by TaSL as a means of identifying GTs that have not started execution; the code that implements it; the GT status, i.e. whether it has committed, aborted, it is executing or has not executed yet and finally a label which is a name given to the GT. The system wide GT id is provided by the Interaction (IA) portion of the CCM as a response to TaSL's request to begin a GT. The label is a name given to the GT by the task definer which is used in different language constructs whose purpose are mainly flow control. These constructs include waiting for a label to be executed in order to take another action; or simply execute label as a means of jumping to the execution of a GT which is defined somewhere else in the program.

(D) Thread data Class. GTs in our model run under threads of control. A thread in this context is a formalism that allows TaSL to support concurrency. When a series of GTs are known to hold no dependencies among themselves, their execution can be interleaved. However, the design of TaSL disallows GTs to interleave their execution if they execute under a common thread of control, for concurrency to be allowed, new threads of control must be forked by the task designer. Thus in the event of one thread of control being rolled back, it is not necessary for any concurrent thread of control to be affected. Data on threads is stored persistently and it is used for reconstructing the tree of execution in case recovery is needed and for maintaining what we call the predecessors list which will be reviewed in the next section.

Among the different pieces of data stored in the thread we have the thread's name, the

parent thread that forked it, the id of the GT executing under it if any, the number of predecessors to the executing GT and a list of the predecessors ids. This predecessor information is used to construct the predecessor table. Each time a GT starts execution under a thread of control the thread data is updated, and a new predecessor list is created. Finally when a thread rejoins its parent thread its data is updated and the thread of control is considered as terminated.

(E) Predecessors Class. Each new GT created forces the update on dependency

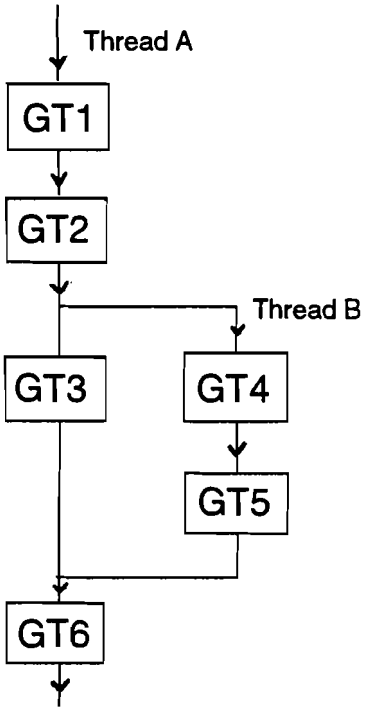

Figure 3.1 : Partial Execution Tree

information. Explicit dependencies' information is maintained in what we call the Predecessors class. In a Predecessor table for each GT executed in the context of an Interaction there exists an entry which points to the list of its predecessors. By predecessors we mean those GTs that executed immediately before in the same thread of control. Despite the fact that only one GT can be executed in a thread of control at a time, a GT can have more than one predecessor, that is its execution could be dependent on more than one GT which is a consequence of allowing multiple threads of control. Consider Figure 3.1, there GT3 and GT5 are considered predecessors of GT6 since GT3 executed immediately before GT6 in the same thread of control and GT5 is a predecessor since it executed immediately before GT6 in a thread of control that joined GT6's thread of control, thus, we consider GT6 to be execution-dependent on GT5.

As expected predecessor information must be maintained persistently throughout the life of the application. Each time a global transaction begins execution the TaSL interpreter via the redirection of the server communicates with the methods provided by the Predecessor class so as to update the information. In the event of a recovery action being initiated due to the invalidation of some committed global transaction the predecessor table must be reviewed and a tree of execution must be built so as to analyze the dependencies and find out which other global transactions if any must undergo recovery.

(F) Label Class. As previously mentioned labels are used as GT identifiers in order to control the flow of execution of a program or to call for the execution of an existing GT. When a label is reached in the process of executing an Interaction, the TaSL interpreter consults with the methods provided by this class in order to investigate whether there exists an execution dependency on the label and if so to take the appropriate action.

The information kept in this class includes, the label itself which is simply a string, the id of the global transaction defined by it and information on whether there exists

some kind of dependency on the label.

(G) Events Class. Events in our work, are as defined earlier conditions on some piece of data belonging to a local database, where the conditions being violated or set is relevant to the execution of the Interaction. Events are of use in the specification of weak conflicts which are set through two constructs analyzed later in the Language. When an event is set due to the specification of a weak conflict the interpreter stores the data relative to the event in the Events Class and notifies the Activator (component of the multidatabase system) of the weak conflict. The Activator in turn watches the database in which the data of the event resides for updates that might violate the condition set forth by the event. In the case of the condition being violated the Activator notifies TaSL. Communication with the Activator regarding events uses event ids that are created by the Activator in response to TaSL's request to declare a weak conflict. The event id together with the weak conflict information is stored in the event class.

In the case of the weak conflict being violated TaSL reviews the event information and takes the appropriate action. Thus, the events class contains the event id, the name of the database in which the event is set and the action or as we call it the handler for reacting to the weak conflict violation. Usually a handler specifies a series of actions that will take the Interaction from an inconsistent state to a valid, consistent one. Event data thus is available at all times allowing the Activator to notify the Interaction of a violation even when the Interaction is not active.

System-Application communication

All data relative to the Interactions is therefore constantly available using the methods provided by the server. The server acts as the liaison not only between the application and its data but between the rest of the system and the applications.

System-Application communication is maintained in two ways. When an application is active the system responds to the applications requests directly through the client in which it runs. For cases where an application either active or inactive must respond to a weak conflict violation or to a recovery process, communication flows between the server and the rest of the System.

Concurrency Control

Finally the server plays the role of Interaction controller. The server maintains a list of the open Interactions and prevents users from opening an Interaction that is being updated by another user. Thus, in reality we disallow concurrency in the update of Interaction, a design decision based on the assumption that users will not update Interactions in a compatible manner.

3.2 The Client

The Client is the only component of the system the users interact with directly. There is a client per active user in the system. At Client start up time which is when the user requests TaSL to start, the client notifies the server of its existence and maintains communication both with the server and the rest of the system throughout its lifetime. A Client consists of two subcomponents: the graphical user interface and the language with its interpreter.

The Graphical User Interface

The graphical user interface is divided into two major components. A global picture of the application and the working environment.

(A) When a user starts up her/his TaSL shell a window with a menu bar and two panels is popped up. This window contains among other menu options,

- New Interaction option: allows the user to create a new Interaction. If this option is selected the shell sends a remote procedure call to the server to investigate if another Interaction with the same name already exists if not the Interaction is created and opened and a window with the working environment is popped up.
- Open Interaction option: Brings up a selection dialog which allows the user to select from the available list of Interactions. Once the user makes a selection the shell brings to the client all the information relative to the Interaction. This information is extracted from our local database through the server. Among all the information brought in we include the tree representing the execution of the Interaction which is displayed in the top panel of the window, in the second panel the state variables with their values are displayed. Our interface design aims at aiding the user in visualizing the state and execution of his/her application.

(B)The working environment: Once an Interaction is opened or created a new window pops up which again contains 2 panels and a menu bar. The two panels are an editor for the user to code his/her Interaction and a display panel for displaying compilation errors and the results of executing GTs. The menu options include, an option for executing an action, an option for aborting a GT, an option for viewing the existing databases and the steps in them, an option for creating a new GT and finally an option for forking and joining threads. Let's quickly review what these options mean in terms of the internal system communication. Note that all communication makes use of the Interaction and GT identifiers.

- Execute Action: The global transaction is interpreted and sent one action at a time for execution to the CCM. The CCM manages the execution of Interactions and their GTs as it deals with concurrency control. From the CCM the global transaction, that is one step at a time is sent for execution to the local level and the results are returned to and displayed by TaSL.

- Abort Global Transaction: The request to abort a GT is passed from TaSL to the CCM who takes charge of it from there and returns the status.
- View of Databases and Steps: TaSL sends a remote procedure call through the server to the methods implemented in our local database to obtain information on the available databases, from there a selection dialog is popped and the user can select to see the steps available in any database. If a selection is made TaSL again requests a step listing by sending a remote procedure call to the local database's appropriate methods. The step listings include a list of the types of arguments that the step takes. Thus the user is allowed to view the available resources at any time.
- New Global Transaction: The working environment window is refreshed and the user can start coding the new GT.
- Forking/Joining Threads: If fork is selected a new working environment is created and all data relative to the new thread is stored via remote procedure calls, through the server into our local repository, and all related information such as predecessors is updated. In the case of a join, TaSL investigates whether the request comes from the parent thread and if so it checks that no GT is active on the joining thread since we do not allow threads to join in the midst of a GT execution. Then, if no GT is executing TaSL stores and updates all related information in the local repository.

Note that as the user creates and executes GTs the execution tree and value and amount of state variables change. Thus for each executed GT TaSL updates the window containing the tree of execution and the state variables by requesting the information from the application data repository.

The Language

The language which we also refer to as TaSL, is used by task definers in modelling the way in which the task will be accomplished. Through the language the task definer establishes the different subtasks, the order in which they will be executed and their

dependencies. It is through the language that access to the local databases is allowed. TaSL is modelled after block programming languages and in the next few sections we will review its features and syntax. TaSL is an interpreted language.

(a) An Interaction is defined by a program which in TaSL it is enclosed in between a *begin_Interaction* $<$ *code* $>$ *end_Interaction*. TaSL however, does not require compilation, it is an interpreted language so a program can start execution without necessarily having been completely written. Such a characteristic is necessary since the kinds of applications that will make use of the multidatabase are not always predictable from beginning to end by the time the definer starts creating the program. This way the task definer has the flexibility to take a new path if the conditions and circumstances under which the task was considered change over time. That is, the task definer interacts with the application.

(b) TaSL allows for the existence of local variables. These variables are necessary for maintaining the Interaction's state.

(c) Actions are the part of the program through which Interaction with the local databases is achieved. An action is defined as a partially ordered set of steps.

(d) Steps are procedure calls. These procedures are defined at the local databases where they actually get executed.

(e) Alternative Constructs: alternatives are used to specify alternative global transactions as well as alternative actions.

(f) Concurrency: TaSL allows the definer to specify which subtasks can be executed in parallel.

(g) Isolation: TaSL through means of some constructs allows the definer to specify when necessary what absolutely cannot happen during the execution of a global transaction.

(h) Weak Conflicts: whenever there exists the possibility of conditions being violated which can invalidate the execution of an Interaction, the task definer can specify a weak conflict which acts like a guard against unwanted effects caused by changes in the local databases.

Next we will review the most important features of TaSL. For a complete syntax specification refer to Figure 3.2.

Interaction Definition

A program is defined in the following form:

$$begin\_Interaction\ \{<Interaction\_definition>\}\ end\_Interaction$$

Variable Declaration

Variables are of great importance in passing information in between different actions relative to the state of the Interaction.

- Form:

  $<type> <name>;$ or
  $<type> <name_1>, <name_2>, \ldots, <name_n>;$

- Possible types: a)standard int, char

  b)date

  c)user defined records of the form:

  record $<record\_name>$
  $<type> <name>;$
  $<type> <name>;$
  ...
  $<type> <name>;$
  end_record;

  Note: records not currently implemented

- Assisgnment: Both the results of a step or an expression can be assigned to a variable:

  $<variable> = <step\_result>;$
  $<variable> = <expression>;$

- Naming: The name of a variable must start with a letter, followed by either letters, digits or the characters "_" and ".".

- Variables in TaSL follow call by value semantics.

Actions

An action is a partially-ordered sequence of procedure calls to steps and it is specified as follows:

$$Step_A \text{ or } Step_B \text{ or } Step_C \text{ or } ... \text{ or } Step_N;$$

with the possibility of specifying a $NULL$ step.

The form for specifying an action establishes the order in which the steps should be tried. As described previously an action succeeds as soon as one of the steps specified in it succeeds. Following the above syntax $Step_A$ will be tried and if successful the action will be considered as completed and no further steps will be tried. In the event of $Step_A$'s failure, $Step_B$ will be tried and so on.

Actions can also be named and stored under that name. In the case of an action being named it can be executed at any time through the *execute* $< action\_name >$ statement. This feature adds power to the language since it allows for reusability of code.

Furthermore, actions can be labelled and placed in the program which allows for the specification of spans of time. That is through this feature the definer can specify that until a label is found a certain condition has to hold or a certain action must be delayed. An action is labelled in the following form: $< label >$: $< action >$ where label follows the naming rules of variables.

Steps

Steps are procedure calls to Step Libraries residing in the local databases. The Step Libraries in turn replace the procedure call with a block of code written in the local database's DML which is then executed locally. The format for a step call is as follows:

$$< LDB >:: < Procedure\_Call >$$

where LDB is the name of the local database that will receive the step for execution and $Procedure\_Call$ is of the form:

$$< Identifier > \ (< argument\_list >)$$

and the $argument\_list$ is as:

$< identifier/constant >$, $< identifier/constant >$,..., $< identifier/constant >$

Alternative Constructs

The specification of alternative actions is done through a variation on the standard if-then-else statement.

$$\begin{aligned} &ifnot \ \{< action_1 >\} \\ &thenifnot \ \{ \ < action_2 > \ \ or \ \ < action_3 > \ \ or \ \ ... \ \ or \ \ < action_m > \ \} \\ &then \ \ ifnot... \\ &then \ \ \{ \ < action_n > \ \} \end{aligned}$$

If an action is non-vital for the execution of an Interaction, then the alternative construct should include the $NULL$ alternative.

Flow Control

Flow Control is accomplished by means of the order of specification. The Flow of a program can be activated or deactivated by means of the wait statement. There can also be change in flow due to weak conflict activation. For all other purposes global transactions will be executed according to their strict order in the program unless

concurrency is allowed. A GT consists of one or more actions and takes the following form:

$$Tran \ < Global\_Transaction\_Specification >$$

where in the case of more than one action being specified, the definer must enclosed their specification in an atomic block, otherwise, each action is considered part of a different GT. An atomic block is specified as follows:

$$atomic \ \{ \ < variable\_declaration >; \ < action\_list > \ \}$$

Concurrency

Flow Control does not necessarily have to happen in a serial manner. There are instances in which a series of subtasks do not hold any explicit dependencies among themselves and can therefore be executed in parallel. To do so the following feature is provided:

$$fork \ < identifier > \ \{ \ < body > \ \}$$

where *identifier* specifies the name of a new thread of control which executes in parallel to its parent thread. The term in parallel however, does not mean that there will be an actual parallel execution, what it means is that the threads can interleave their execution in any manner because there exists no explicit dependencies. The *body* in the above syntax specifies the series of GTs executing under the new thread. Once the new thread of execution completes all of its subtasks it is ready to be joined with its parent thread, and only its parent thread can call for the join. The join statement is of the form:

$$join \ (\ < identifier\_list > \ )$$

where the *identifier_list* could be any number of threads being joined at once.

Isolation

Isolation is specified through the atomic construct described above. A series of actions enclosed in an atomic block specify that no other subtask containing conflicting operations can interleave its execution with the subtask specified in the atomic block.

Events and Waiting

An event is specified as:

$$< LDB >::< Identifier >, < Condition >$$

where the condition can be an algebraic expression which results in a boolean value.

A wait statement is used to control flow of execution. It is of use for cases where the Interaction must wait for some condition to be true in order to continue execution. Example: An airplane reservation must be confirmed before a hotel reservation is made. The wait statement has the following form:

$$wait\ (< event >)$$

Weak Conflicts

Weak conflicts are used as means of determining positive or negative changes in some database which affects the execution of an Interaction. It is weak conflicts what makes Interactions reactive. A weak conflict being set implies a possible change in the execution of the Interaction. For example: if our weak conflict states that in the case of our plane reservation being cancelled to make reservations with another airline, the Interaction changes route as the change of reservations might affect a chain of other subtasks. The effects of a weak conflict being violated might set forth the re-execution of part or all of the Interaction in which it was set. A weak conflict thus might trigger the rollback of a series of subtasks that conform an Interaction. In this case, TaSL

must be prepared to communicate with the Recovery Manager of the multidatabase in order to design the recovery process. As we will see in section 3.3 the design of the recovery process will include deciding which of the subtasks which exhibit some kind of dependency on the subtask being rolled back actually need to be rolled back. For example: the hotel reservation does depend on the flight reservation, however, a change of flight reservation does not necessarily imply a change in the hotel reservation as long as the state in which the new flight reservation leaves the Interaction is no different than the state the hotel reservation encountered the first time around.

We placed as requirement on weak conflicts, that they be set within the GT that originates them. Such restriction responds to the possibility of the weak conflict being violated in the window of time between the completion of the GT and the setting of the weak conflict which would in turn cause the Interaction to be unaware of the violation. The form of a weak conflict is as follows:

$$[\ \textit{until} < \textit{label} > \ ]\ \textit{on}\ \ < \textit{event} >\ \ \{< \textit{compound\_statement} >\}$$

where the square brackets indicate optional statements. The *compound statement* establishes a a handle which specifies what needs to be done.

```
<interaction> := begin_InterAction {
                 <Declarations>
                 <Body>
                 } end_InterAction
<Declarations> := <Declarations> <Type> <Identifier_list>; | ε
<Type> := int | char | date
<Identifier_list> := <identifier >, <Identifier_list> | <identifier>
<Body> := <Body>; Tran <identifier> atomic {
          <Declarations> <compound_statement> } |
          <Body>; Tran <identifier> <statement> | ε
<compound_statement> := <statement_list> | ε
<statement_list> := <statement_list>; <statement> | <statement>
<statement> := ifnot { <statement_list> } then { <statement_list> } |
               ifnot <statement> then { <statement_list> } |
               ifnot { <statement_list> } then <statement> |
               ifnot <statement> then <statement> |
               <action_statement> | <assignment_statement> |
               <wait_statement> | <abort_statement> | <fork_statement> |
               <join_statement> | <weak_conflict_statement> |
               <execute_statement> | NULL
<action_statement> := <step_list> | <label>: <step_list>
<step_list> := <step_list> or <step_call> | <step_call> | NULL
<step_call> := <identifier> :: <identifier> (<argument_list>)
<argument_list> := <argument_list>, <identifier> | <identifier> |
                   <argument_list>, <intlit> | <intlit> |
                   <argument_list>, <string> | <string> |
                   <argument_list>, <datelit> | <datelit> | ε
<assignment_statement> := <identifier> = <step_list> |
                          <identifier> = <expression> |
                          <identifier> = <string>
<expression> := <simple_expression> |
                <simple_expression> <comp_op> <simple_expression>
<simple_expression> := <term> | <sign> <term> |
                       <simple_expression> <add_op> <simple_expression>
<term> := <factor> | <term> <mul_op> <factor>
<factor> := <variable> | <intlit> | ( <step_list> )
<variable> := <identifier>
<wait_statement> := wait ( <event> )
<event> := <identifier> :: <identifier>, <condition>
<condition> := <comp_op> <variable> | <comp_op> <intlit> | delete
```

Fig3.2 : Formal Definition of TaSL

$<abort\_statement> := abort\ <label>$
$<fork\_statement> := fork\ <identifier>\ \{ <Body> \}$
$<join\_statement> := join\ (\ <identifier\_list>\ )$
$<weak\_conflict\_statement> := until\ <label>\ on\ (\ <event>\ )\ \{ <Body> \}\ |$
$on\ (\ <event>\ )\ \{ <Body> \}$
$<execute\_statement> := execute\ <identifier>$
$<label> := <identifier>$
$<datelit> := <intlit>$
$<string> := <charlit>$
$<identifier> := [A-Za-z]+\ \{\ [\_\ |\ .]\ |\ [A-Za-z]\ |\ [0-9]\ \}$
$<intlit> := [0-9]+$
$<charlit> := [A-Za-z]+$
$<sign> := +\ |\ -$
$<add\_op> := +\ |\ -$
$<mul\_op> := *\ |\ /$
$<comp\_op> := <>\ |\ <\ |\ >\ |\ <=\ |\ >=\ |\ ==$

Fig3.2 : Formal Definition of TaSL (Cont.)

3.3 Recovery

The Recovery Manager and TaSL keep very close links, since information flows both ways. In what follows we will take a quick look at the kinds of issues that the RM and TaSL confront as they work towards maintaining application validity.

Compensation is the primary means of recovery. In order to decide which GTs need to be compensated for, or otherwise aborted, when part or all of an Interaction becomes invalid, the RM must recreate the tree of execution and dependencies from the GT that causes the invalidation down to the last executed/executing GTs. Note that GTs in this dependency tree which have not committed are aborted. The Recovery Manager has information on the explicit dependencies, i.e. on execution dependencies. The CCM informs the RM of all executed transactions which the RM logs. However, as we mentioned earlier in this paper there exists yet another kind of dependency, which I call an implicit dependency, of which the RM has no knowledge. Implicit dependencies are caused by the reading/writing of state variables among different GTs. That is, consider two GTs in very different execution paths, i.e. executing under different threads of control. It is enough for one of them to read from a variable that the other one wrote to for an implicit dependency to be created. This kind of dependency must also be taken into account in the recovery process, otherwise, we run the risk of leaving the Interaction in an invalid state even after the recovery takes place. Example: Consider that GT6 running under thread A writes to a variable x which is read afterwards by GT12 running under thread B. Now further suppose that the two threads do not join execution in the span of time we are concentrating on. Now, let's assume, that after GT6, GT7 and GT8 committed under thread A and GT9 is currently executing. At this point a weak conflict set forth by GT6 is violated, placing the Interaction in an invalid state. Let's assume that the conflict states that GT6 must be compensated for and redone. The RM creates the execution tree from the explicit dependencies and finds that GT9 must be aborted, GT8, GT7 and GT6 must be compensated for in that

order, and finally, GT6 must be redone. However, GT12 had read from x a value which after the compensation and redoing of GT6 might no longer be valid. In our system we took these dependencies into account, and we solve this issue by keeping the RM in close communication with TaSL. Thus, the RM is able to derive the complete tree of dependencies by gathering the implicit dependencies information from TaSL.

Furthermore, although the RM directs the Recovery process, the redoing of GTs happens through TaSL. That is, when the RM wants to reexecute a GT, it calls on TaSL to manage the execution. Note however that the redone GT is initiated by the RM.

Communication between TaSL and the RM is also vital for the good development of an application. If TaSL were not informed of aborted GTs or compensated GTs the user would not be informed either, which would imply that what TaSL and the user believed as accomplished tasks of the application might no longer be there. Thus, we would have an inconsistency between what the application has actually accomplished and what TaSL and the user believe it has accomplished.

Finally, the RM is designed to optimized the recovery process. This optimization is based on the idea that it is not always necessary to compensate or rollback all GTs dependent on the invalidating GT. Let's consider the example where the invalidating GT consisted of making a flight reservation. The task definer had set forth a weak conflict which established that in the case of the reservation being cancelled the execution of the Interaction should be retried from the invalidating GT. That is, a new reservation should be made and all dependent GTs should be retried. Now, assume that one of the dependent GTs was in charge of making a hotel reservation and its only dependency on the invalidating GT was the day the flight was arriving at the destination. Now if the invalidating GT is redone and a reservation is made which places the customer at the destination in the same day, there exists no apparent reason for compensating for the hotel reservation and later redoing it. That is, the GT in charge of the hotel reservation would be reading and receiving the same state as it did the first time it executed. The

only way the RM can find out the task definer's intentions and the actions that will be taken after recovery as well as the actual state affecting dependent GTs is by maitaining communication with TaSL. TaSL provides the RM with a global view of what will happen after recovery and how the value of state variables after affect the validity of dependent GTs. Thus, the RM is able to apply its optimization by determining what should and should not happen after compensation as well as establishing how the state after reexecution of the invalidating GT affects or doesn't the validity of dependent GTs.

REFERENCES

[ANRS92] Mansoor Ansari, Linda Ness, Marek Rusinkiewicz, Amit Sheth. Using Flexible Transactions to Support Multi-system Telecommunication Applications. In *Proceedings of the 18th VLDB Conference*, pages 65-76, 1992.

[ELLR90] A.K. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A multidatabase transaction model for InterBase. In *VLDB Proceedings*, pages 507-518, 1990.

[GS87] Hector Garcia-Molina and Kenneth Salem. Sagas. In *ACM SIGMOD Proceedings*, pages 249-259. ACM, 1987.

[MN91] Marian H. Nodine. InterActions: A non-serializable global transaction model for heterogeneous multidatabases. Technical Report CS-91-64, Brown University Department of Computer Science, December 1991.

[MNF92] Marian H. Nodine. Supporting Planning Tasks on an Evolving Multidatabase Using InterActions and Events. Report, Brown University Department of Computer Science, February 1992.

[MND92] Marian H. Nodine. Interactions: Multidatabase Support for Planning Applications. Report, Brown University Department of Computer Science, December 1992.

[OS] ObjectStore is a product of Object Design Corp.

[WR91] Helmut Waetcher and Andreas Reuter. The Contract model. In A. Elmagarmid, editor, Database Transaction Models for Advanced

Aplications. Morgan-Kauffman, 1991.