# Class Library Support for Workflow Environments and Applications

M. Papazoglou, *Senior Member*, *IEEE*, A. Delis, *Member*, *IEEE*, A. Bouguettaya, *Member*, *IEEE*, and M. Haghjoo

**Abstract**—Workflow systems are receiving increased attention as they intend to facilitate the operations of enterprises by coordinating and streamlining business activities. The need for automated support and operational models that allow workflow applications to coordinate units of work across multiple servers—according to business defined rules and routes—is becoming critical for the proper management of such activities. In this paper, we describe a Transaction-Oriented Workflow Environment (TOWE) for programming workflow activities. The novelty of our approach resides in the proposed unified abstraction, *class libraries*, to support workflow activities. The fundamental concept used in the TOWE system is based on the symbiosis of object-oriented programming and interprocess communication concepts. In TOWE, the concurrency abstractions are represented by *process objects*, active objects acting as processes, which involve asynchronous, location-independent, and application specific process invocations.

**Index Terms**—Workflow systems, multidatabase systems, object-oriented programming, class libraries, distributed systems, nested transactions, process objects, concurrency, scheduling and synchronization.

## 1 Introduction

The increased use of powerful workstations and high-bandwidth networks has motivated enterprises to move toward distributed computing environments to support information exchange and collaboration between users. There is an increasing demand for the coupling of local applications with applications running on remote servers to support greater workgroup and organizational productivity. Business applications requiring this kind of support include airline and hotel reservations, health-care, banking and finance, stock-brokerage, manufacturing, and telephone directory services. The ultimate goal is to provide a richer information creation and analysis environment for the user by integrating the user's desktop facilities with an information exchange and collaboration infrastructure including mail, messaging, groupware platforms, shared databases, workflow systems, and document servers.

Workflow applications are receiving increased attention as they intend to automate the movement and execution of a number of units of work, across one or more servers, according to business defined rules and routes. A workflow typically defines the individual business activity steps, the order and the conditions under which the activities must be executed, the flow of data between activities, the users responsible for the conduct of these activities, and the tools used. Transaction support for such cooperative applications is inherently more complex to program than applications which rely on short and noninteractive transactions. Network-centric systems, such as workflows, require concurrent access to shared and mutable data originating from disparate information sources. They, therefore, demand nontraditional and rather complex transaction mechanisms [15]. Workflow environments require sophisticated (nonconventional) transaction mechanisms to support the sharing of uncommitted data between concurrently active subtransactions. These requirements are mainly due to the collaborative and distributed nature of workflow applications and can also be found in other data-intensive applications such as office automation computer-aided design, and computer-aided software engineering (CASE) applications.

Until recently, the traditional approach to distributed transaction management relied on variants of the *closed nested transaction model* [23] to manage remote transactions. This model introduces limitations as it adheres strictly to the classical serializability paradigm to control network-wide transaction management and provide full isolation at the global level. Several extensions to the conventional closed nested transaction model—collectively referred to as *open nesting*—have been proposed to increase transaction concurrency and throughput, by relaxing the atomicity and isolation properties of the traditional transaction model [30]. Such efforts have mostly focused on flexible transactions for multidatabase systems that provide mechanisms for the specification and flow of control among different work units. However, most of these efforts to date have only concentrated either on conceptual transaction models, concurrent versions of algorithms and theoretical results on model correctness. Moreover, flexible transaction models which have adopted a data-centered approach are rather strict and lack proper implementations. In summary, relatively little attention has been given to software development

- *M. Papazoglou is with Tilburg University, INFOLAB, Room B304, P.O. Box 90153, 5000 LE Tilburg, The Netherlands. E-mail: mikep@kub.nl.*
- *A. Delis is with the Department of Computer and Information Science, Polytechnic University, Brooklyn, NY 11201. E-mail: ad@naxos.poly.edu.*
- *A. Bouguettaya and M. Haghjoo are with the School of Information Systems, Queensland University of Technology, GPO Box 2434, Brisbane, Qld 4001, Australia. E-mail: {athman, mostafah}@icis.qut.edu.au.*

*For information on obtaining reprints of this article, please send e-mail to: transcom@computer.org, and reference IEEECS Log Number 104647.0.*

environments that are required to convert these efforts into operational distributed application programs.

In contrast to flexible transaction models, workflow management systems, due to their very nature, are not data-centered. Their aim is to provide modeling support and mediate communication, interaction, and coordination among collaborating people and business activities within and between organizations. Such activities are better supported by a process-centered approach. Our work addresses this situation by providing unified programming and distributed system support necessary to program network-centric workflow applications. These combined facilities come in the form of class library facilities that implement transaction extensions of an object-oriented programming language coupled with distributed system support. For this purpose, we developed a Transaction-Oriented Workflow Environment (TOWE). TOWE provides facilities for the construction and programming of long-lived concurrent, nested, multithreaded activities. Our approach to programming long-lived activities follows the *class library based approach* [6], [3] by building class libraries from a small set of fundamental concepts that are extensions and refinements of open nested transaction constructs. This environment enables the design of flexible distributed applications by providing library modules that represent abstractions of the various system aspects and functionality of long-lived activities in a concurrent object-oriented environment. It also supports the reuse and specialization of existing workflow specifications.

This paper is organized as follows: Section 2 provides an overview of the related work, while Section 3 gives an overview of the TOWE. Section 4 discusses programming of long-running activities in TOWE. Section 5 discusses our experience with the current implementation. The conclusion and future work can be found in Section 6.

## 2 Related Work

Traditionally, workflow systems have been used as a way to describe office functions and procedures. Such descriptions were normally based on extensions of well-known formal models, such as Petri-nets, production rules, and flow-charts [33]. The need for a closer interaction between workflow and process modeling techniques in software engineering was also identified, resulting in several workflow methodologies [21] and specification languages [8]. Our design is in line with these suggestions as it combines the functionality of a modern object-oriented language with distributed message-passing system support. This enables a collection of independent information repositories to be programmed and used as a coherent and flexible concurrent computation resource.

Script-based sublanguage approaches, such as ConTracts [34] and Interactions [24], are also closely related to this work. ConTracts were proposed as a mechanism for grouping transactions into a coherent unit of work. ConTracts are scripts of steps, where steps are mapped into transactions on local databases. ConTracts provide mechanisms for relaxed atomicity and isolation by controlling the flow of long-lived activities. Interactions are open nested flexible transactions used to define long-lived tasks that access multiple databases. This system supports the ability to backtrack when a transaction aborts and causes its subtasks to fail. Other projects with common aims which offer specialized script languages for the execution of workflow models driven by intertask dependencies include the METEOR [17] and the MOBILE [31] projects. Sublanguages offered by the previous approaches are only limited languages in that they provide only a handful of special purpose primitives and control mechanisms mainly tailored around open-nested transactions. Such linguistic facilities are modeled primarily after block-structured programming languages, e.g., in ConTracts and Interactions.

Other activities on workflow specification and management related to our work, such as the Mentor project [36], place emphasis on deriving a form of a distributed workflow from a formal specification. Techniques are proposed for transforming a centralized state chart specification into a form that is amenable to a distributed execution of workflow activities. A different approach is suggested in [16]. This work concentrates on workflow specification techniques based on Event-Condition-Action (ECA) rules, as used in active database systems [35], for describing the control flow between tasks. Dependencies between tasks are expressed by events and emphasis is placed on prespecifying the task execution capabilities of individual problem solving components. In contrast to these approaches, our work places more emphasis on providing the run-time environment on which workflow processes are coordinated and executed based on concurrent object-oriented programming support for transactional workflows. The TOWE services can be used as a vehicle to implement higher-level concepts and mechanisms, such as those proposed by the above two projects.

On the office automation front, our work presents some similarities with the work reported in [37] and [19]. In [37], a distributed office information system is proposed. Two types of cooperative office activities are identified. The first type describes office activities that are *structured* while the second type describes activities that are *unstructured*. The publication concentrates on unstructured activities. It provides a conceptual framework based on a modified object-oriented model and makes use of a distributed knowledge base system implemented as *logical workstations*. Our work differs from this work on several grounds. This approach remains mainly at an abstract level and does not propose an operational model. In contrast, our work focuses on structured activities and proposes an extensible class based approach that is coupled with a distributed multithreaded communication protocol for cooperative work. More importantly, in [37], there is no indication as to how cooperative tasks are to be supported. This is probably due to the unstructured type of tasks that this work is focused on. In [19], a programming environment called OASIS is described. It allows for the management and coordination of micro-organizational activity processor (MOAP). In contrast to our work, this system focuses on the use of knowledge base systems for defining and managing micro-organizations and does not address any cooperative problem solving activities.

On the programming language front, there is a strong correlation between the notion of modularization in modern programming languages and transaction nesting. There have been some attempts to capitalize on this relationship, most notable of which is Argus [18]. Although Argus provided linguistic support for implementing the execution of distributed programs based on the closed-nested transaction philosophy of Moss [23], it lacked the proper database and operating system support, and permits only a single thread of control to execute within one transaction. These limitations have been addressed, at least partially, by the work of Haines et al. [13] which describes a modular language for supporting transactions based on high-order functions. This work proposes a general purpose control abstraction mechanism through which one can compose flexible transactions.

Our work differs from the above activities by providing a richer variety of semantic-oriented transaction primitives in the form of an extendible/specializable class library combined with object-oriented language and distributed system support. More importantly, we provide a full *linguistic* framework which borrows some properties, such as *futures* and *continuations*, from the Actor Model [1]. We also provide an open system to support long-lived transactions in a workflow environment.

## 3 The Workflow Programming Environment

Workflow systems can be viewed as containing two interrelated components [8]: a *specification* and an *execution* module. The prime concern of the specification module is to provide modeling support in a way that enables administrators and analysts to define business-related procedures and activities, analyze and simulate them, and assign them to people. Usage of this module is typically completed before the flow of work tasks commences. The workflow execution module is the component which interacts with an *invoked application interface* seen by programmers and end users. Its purpose is to enable the workflow engine to activate an application to undertake a particular work unit by providing the proper execution environment. It achieves this by supporting the coordination and conduct of business procedures and activities. The workflow execution module enables the work units to *control* the flow from one workstation to another as the steps of a business activity are carried out. The work described herein concentrates only on the workflow execution environment and does not consider such issues as automated workflow modeling or assignment of people to work units.

In this paper, we view a workflow as a long-lived activity that coordinates the execution of multiple process-oriented tasks (with transaction properties). These tasks have interrelated dependencies capturing both data and control flow requirements and constraints. A long-lived activity in TOWE is divided into a number of work units, nested to multiple levels, which may execute sequentially or concurrently. Work units that can be found only on the leaf-level of an activity tree are referred to as *actions*. Actions are a form of process, some of which may execute at remote (*component* database) sites where they are mapped to native transactions. The intermediate nodes in a workflow hierarchy are normally compound actions or *intermediate activities* which comprise a set of elementary actions. To indicate the process nature of activities, actions, and compound actions, we frequently use the word *process* to describe them.

### 3.1 Functional Model

An intermediate activity is a unit of action scheduling that corresponds to the specification of a state of execution within a top-level activity. Actions are atomic units of work mapped to counterpart flat component database transactions that support the conventional ACID (atomicity, isolation, consistency, and durability) transaction properties. Consequently, the interface of a component transaction system provides, at least, the following operations: `begin`, `prepare`, `commit`, `abort`, and `end` transaction. It is an important feature of TOWE that every action runs under the protection of a classical transaction with ACID properties and that several of these actions can be combined to run inside the boundaries of the same activity.

Fig. 1 gives a top-level view of an activity initiated by a database site in a distributed client-server environment. Activities in this environment are seen to correspond to a multilevel tree. The root of the tree represents the top-level activity itself (level-0), the actions spawned by a top-level (or intermediate) activity correspond to exported high-level operations (by the particular site on which the action executes) and reside at level-1. Finally, level-2 corresponds to native transaction operations executing at these remote sites. In this figure, we assume that the distributed workflow application was initiated at the left-most site (client) and runs on a number of sever databases.

In the following, we will concentrate only on level-0 and level-1 and will not consider the mapping of actions to native transactions. Such activities are considered in a companion paper [27]. In [31], the authors discuss the use of similar mechanisms such as Transaction Processing (TP) monitors and Object Request Brokers (ORBs) to construct object wrappers for workflow-unaware application programs. For reasons of simplicity and clarity, we take a different approach by assuming that each site is equipped with an object-oriented wrapper that maps exported operations (TOWE *aliases*) to equivalent native operations (Fig. 1). For a detailed treatment of how object-oriented wrappers act as intermediaries between native and remote systems, we refer the reader to [2].

### 3.2 Flow of Activities

Transactional workflows are used to automate business systems that have definable, repetitive, and well understood policies and procedures. For example, a mortgage loan approval application is a well understood process that goes through a prescribed set of procedures. The processing of a loan goes through standard phases regarding the processing and routing of intermediate units of work. A mortgage loan plan is a long-duration activity which may span from hours to several days. This application must outline specific procedures and protocols to be carried out, preserve ongoing work, coordinate inter-related pieces of

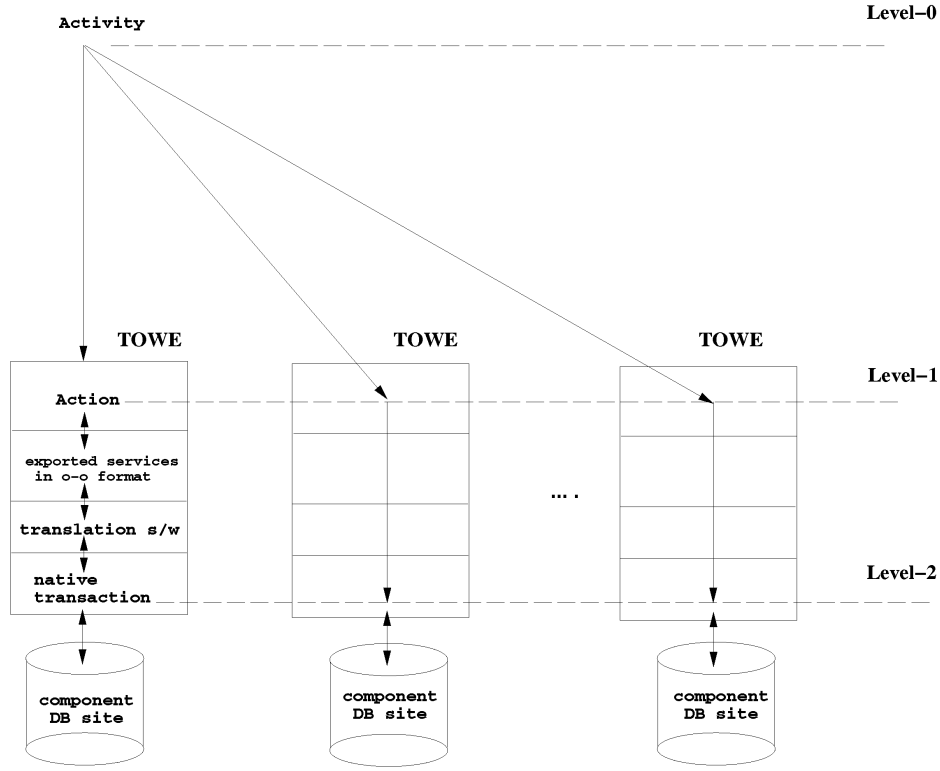



Fig. 1. Top level view of long-lived activity.

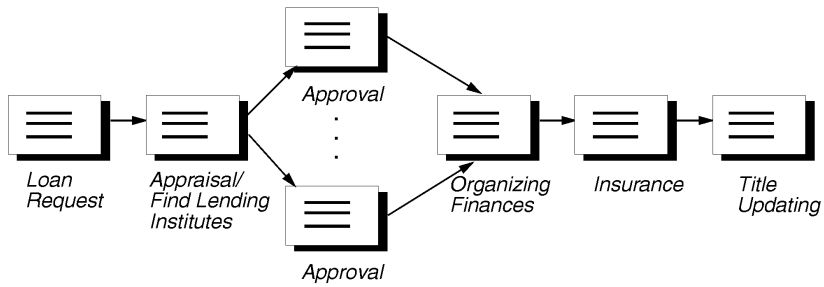



Fig. 2. Workflow view of a mortgage loan.

work, respond to business events, schedule further work, and so on. The flow of events for this situation is depicted in Fig. 2. A mortgage loan plan is a form of a workflow because it describes the steps for dealing with a loan request.

To illustrate some of the workflow specification features of the TOWE, we use a sample application program that models a mortgage loan planning activity according to the graphical notation depicted in Fig. 3. This notation models a workflow activity as an acyclic directed graph in which leaf nodes (ovals) represent actions; nodes represent intermediate activities and are symbolized by either rectangles or pentagons; and, finally, edges represent the flow of control while specialized arc symbols represent the flow of data and synchronization modes between simple and compound actions. Fig. 3 employs an *activity tree* that represents the events described in Fig. 2 by indicating the type of operations a unit of work traverses and the routes that specify acceptance conditions for moving from one operation to the next.

In Fig. 3, we assume that a mortgage loan application seeks loan approval from a number of lending institutions (shown as branching approval activities in Fig. 2) which may potentially finance a customer purchasing a residential property. The application tries to first secure a home loan by obtaining funding from banks (in terms of preferences specified by the customer) and then tries alternative funding sources, e.g., credit unions. This involves interaction with funding institutions in geographically distributed locations and is shown by means of a high-level activity called **`Appraisal`** which spawns two processes **`Find_Bank`** and **`Find_Credit_Union`**. We assume that this activity can execute its units of work in parallel and can succeed if both of its children succeed, irrespective of which one finishes first. The **`Find_Bank`** process is a serial alternative activity which means that one bank is tried out after the other until a condition is satisfied, whereas **`Find_Credit_Union`** is a parallel alternative activity which means that many credit-unions are tried at the same time and we select the first one which responds positively. If the customer is successful in securing funds, then the serial activity **`Approval`** is executed. This process forks a series of sequential processes which open a

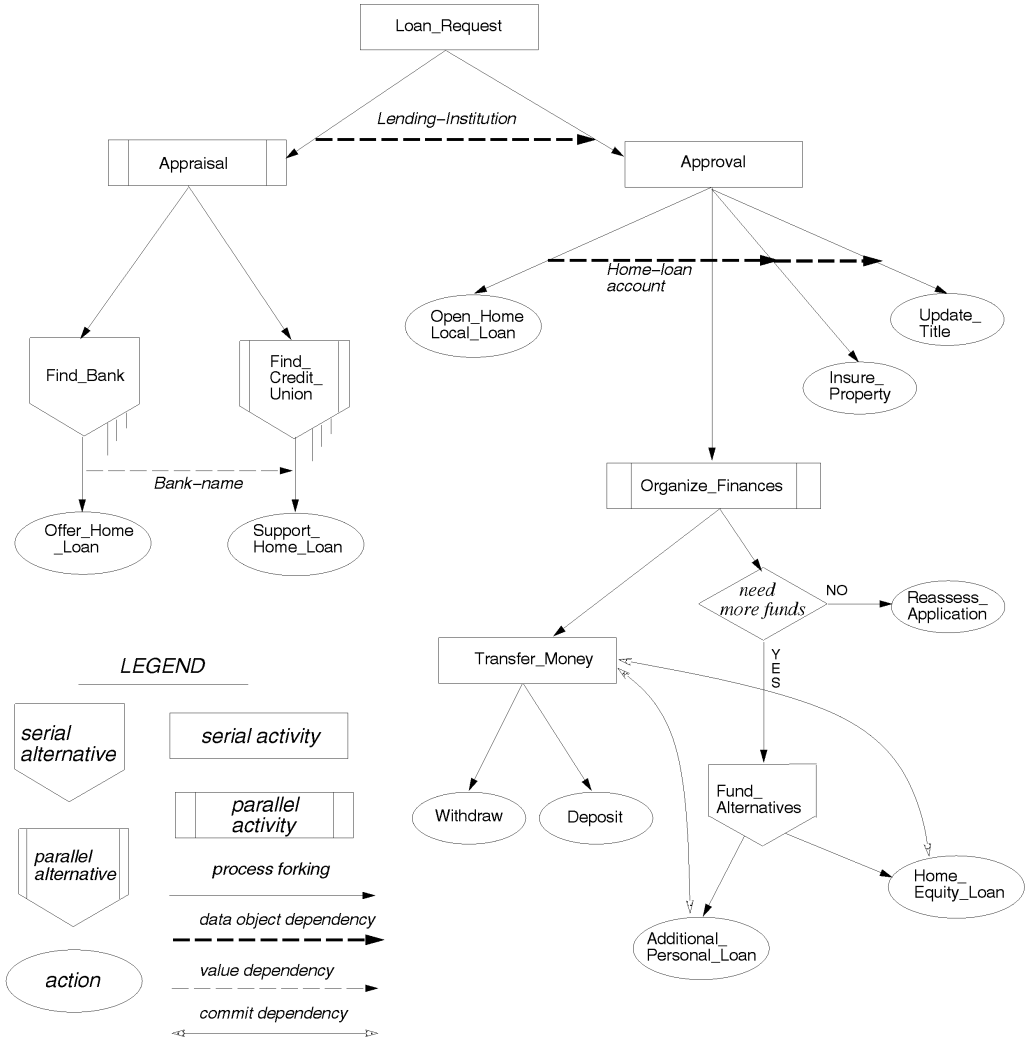



Fig. 3. Activity tree for a home loan application with intra and interdependencies.

home loan account first, organize finances, insure the property (only if the activity **`Organize_Finances`** has succeeded), and finally update the title. Attention is drawn to **`Organize_Finances`** which is a parallel activity. Its first process transfers funds from remote account(s) to the home loan account to buy a residential property. Its second process is the conditional connector **`More_Funds_Needed`** which detects situations where a customer needs more funds to cover other expenses, such as stamp duty, solicitor fees, etc. In this case, other options may be attempted, e.g., a personal loan or a home equity loan (a line of credit secured by a registered mortgage over a residential property).

### 3.3 Scheduling and Synchronizing Processes

The flow of control and data between the work units of an activity needs to be explicitly specified and scheduled. Scheduling is required since the actions running on remote databases access and change shared data items and require synchronization to avoid corrupting these data items. If actions fail, their effects must be retracted and other actions in other remote databases which have "observed" their effects must also be retracted. To achieve these effects, TOWE relies on the existence of a *scheduler* process.

Our design unites the constructs of class and process types as they both support local variables, persistent data, encapsulated behavior, message passing mechanisms, and restrictions on how modules exchange information [22]. Our design provides scheduling library classes which run on each component database. Instances of specialized scheduler classes are created for every long-lived activity. This allows the construction of concurrent functions to be performed separately from task scheduling and leads to increased concurrency in application programs [12].

In TOWE, the concurrency abstractions are represented by *process objects*, a type of active objects acting as processes, which involve asynchronous, location-independent, and application specific process invocations. All scheduling and synchronizing objects are process (active) objects. These should be distinguished from passive objects which are the TOWE *data objects*, such as **`Accounts`**, **`Customers`**, **`Property`**(ies), various types of **`Loan`**s on which the process objects operate.

We distinguish between four types of scheduler processes, implemented as scheduling classes, which are sufficient to program the various types of activities in a workflow application.

1) A *serial scheduler* whose actions are submitted and committed sequentially, e.g., the **Approval** activity in Fig. 3. Serial schedulers establish a *begin-on-commit* dependency with each other, i.e., one cannot begin unless the previous one commits.

   All actions should commit in order to make their parent transaction commit. If any of them fails, their parent transaction aborts including all of its temporarily committed descendants. This type of abort mechanism is based on a single specialized compensating transaction which *undoes* temporarily updated data items from a cancel-log to bring them to their original values, see Section 4.3.

   Actions of a serial scheduler may have *data object dependency*. Data object dependencies correspond to passing an argument in the form of a referenced object between consecutive actions, i.e., one action passes the state of an entire data object to another action, for the second transaction to execute. This is indicated by the presence of a bold dashed arrow in Fig. 3 where both actions **Insure_Property** and **Update_Title** need the data object **Home_Loan_Account**.
2) A *parallel scheduler* which allows all of its actions to be submitted and executed in parallel as independent activities. These actions also commit independently. It is expected that all actions should commit before their parent commits. Such actions may establish two types of dependencies:
   - *Value dependency*, this situation corresponds to a receiver blocking and awaiting for a specific attribute value to arrive before it starts executing. In this situation, we are expecting to receive only a single attribute value rather than the entire repertoire of attribute values for a given data object. This type of dependency should be contrasted to the data object dependency (described above) where we expect to receive the entire object state rather than only a single attribute value. This blocking may occur at some point when the receiver needs a particular attribute value produced by a sender process. At this point, both the sender and receiver are concurrently active. Refer to the case of **Find_Bank** and **Find_Credit_Union** processes where the latter needs a bank-name to complete its computation.
   - *Commit dependency* when two activities may go in parallel but one may not commit unless the other commits first. This is indicated by the shaded arrows in Fig. 3, where the actions **Additional_Personal_Loan** and **Home_Equity_Loan** cannot commit unless the activity **Transfer_Money** has committed first, see Fig. 3.
3) A *serial-alternative scheduler* is a scheduler which attempts actions sequentially until one produces the desired outcome, e.g., the **Find_Bank** process in Fig. 3. The parent only aborts if all its descendent actions were tried unsuccessfully or if the transaction is timed out.
4) A *parallel-alternative scheduler* where alternative choices are pursued in parallel. This is the case with the **Find_Credit_Union** process. As soon as any one of the actions succeeds, the scheduler commits and the effects of all other parallel actions are discarded.

Schedulers may also have *conditional actions* which get activated if certain conditions hold at run time, see the activity **More_Funds_Needed** in Fig. 3. They may also have *replicated actions*, where the same action (or activity) can be sent to different sites in terms of a parallel alternative scheduler. Examples of application code developed around the above scheduling processes can be found in Section 4.

### 3.4 Language and System Support

To develop TOWE, we targeted an appropriate programming language and opted to add new capabilities to it in the form of programmable library classes. For this purpose, we interfaced the object-oriented language Sather [25] with the widely available distributed interprocess communication package PVM (Parallel Virtual Machine) version 3.0 [10], which is responsible for distributed programming and message passing.

Sather is a *core* object-oriented language derived from Eiffel and places particular emphasis on writing efficient and reusable code. It provides appealing object-oriented features which make it adaptable to the programming of distributed workflow applications. Such features include parameterized classes, separation of specification from implementation, statically-checked strong typing, high-order routines, exception handling, pre- and post-conditions, and class invariants [25]. PVM, on the other hand, consists of a collection of protocols that implement reliable and sequenced point-to-point data transfer, message broadcasting, process control, barrier synchronization, and mutual exclusion [10].

To support the specification of distributed applications, the TOWE provides a collection of programmable library classes that can be extended and specialized according to the application's needs. The TOWE language provides an interface component which defines the callable methods which are based on PVM primitives. A site may specify available remote services in the context of specific classes and the TOWE provides the facilities which make the execution of remote calls at these sites possible. The TOWE language supports strong typing and static type checking to ensure type safety. It is also *object preserving* in that it returns objects of existing types to preserve data integrity.

In addition to the general purpose programming features described above, the class library comes equipped with other salient features, which make it particularly suitable for the programming of long-lived activities. These support:

1) Specification of *chronological (temporal) dependencies* among activities, thereby allowing more flexibility for database programmers in transaction scheduling.
2) Specification of *value* and *data object* dependencies for all types of concurrent activities and actions including actions activated at different points in time without prior run-time knowledge of each other.
3) *Exception handling* mechanisms are provided to execute predefined handlers depending on the type of exception raised, in cases where a failure occurs during

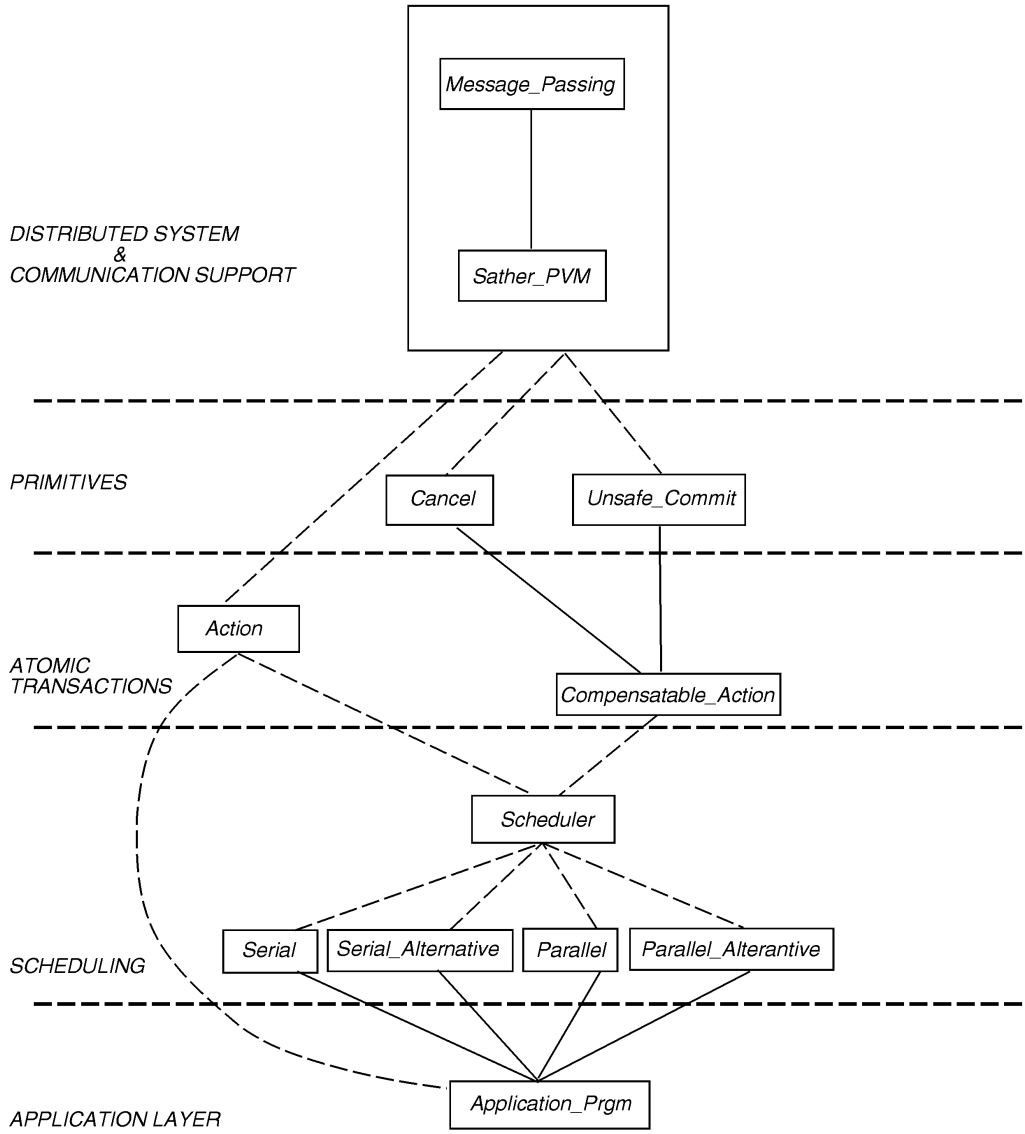



Fig. 4. Library classes of the TOWE.

execution of an action. This allows us to trap and distinguish different kinds of aborts. Exception handlers are normally implemented by invoking compensating actions.

4) Specification of *commit dependencies* between actions so that a task waits for a signal from another, i.e., blocks, before it is allowed to commit. Also, automatic *cancel* procedures are provided to semantically undo the effects of unsafely-committed actions if the global activity fails.
5) Specification of *compensating actions* are used to undo, from a semantic point of view, the effects of an action at a particular site. Rules defining compensating transactions are *attached to objects* and facilities are provided for transactions to distinguish between such tasks and proceed accordingly.
6) Specification of *contingency actions* to execute in case that a given transaction fails. Actions can be *vital* or *nonvital*. If a vital transaction aborts, then its parent must abort. Nonvital actions can be simulated by serial-alternative and parallel alternative schedulers, whereas serial and parallel schedulers can simulate vital transactions.
7) Finally, the class library allows for added flexibility by supporting *ordinary messages* which are dispatched at the end of the receiver's message queue and are buffered; and *express messages* (signals) which get processed right away no matter whether the receiver is active or has other messages in its queue.

In TOWE, *temporal dependencies* are usually implemented using both intra and interactivity dependencies. For instance, a *serial schedule* dependency is a temporal dependency between serial activities (e.g., **`Loan_Request`**). An example of an intraactivity dependency that models a temporal dependency is the *value dependency* (e.g., value dependency between **`Find_Bank`** and **`Find_Credit_Union`** where the synchronizing value is *Bank-name* within the activity **`Appraisal`**). Contingency actions may also establish a form of *temporal dependency*. For example, if an activity

such as `Find_Bank` cannot be completed within its timing constraints, then we may decide to execute another activity such as `Find_Credit_Union`. Hence, we establish a time dependency between these two activities.

The TOWE supports the features described above and supports typical data transfer facilities such as point-to-point and selective broadcast. Interprocess communication is implemented via message passing. Activities and actions can be initiated synchronously or asynchronously and may be conditioned upon the initiation or termination of another action / activity.

Synchronization between actions belonging to different families are handled by the system in a manner analogous to that of Camelot [9]. Synchronization between processes executing in different sites is provided by means of mutual exclusion and synchronization primitives such as mutexes, condition variables, blocking and nonblocking receives, and barrier constructs.

### 3.5 The Class Library

The TOWE library is a collection of specialized classes implementing the primitives and constructs described in Sections 3.3 and 3.4. A distributed application is developed by creating instances of the library classes.

Fig. 4 depicts the TOWE library classes that act as abstract data types providing a basic set of operations which can be further specialized depending on the needs of the applications. These classes are organized in five broad categories which range from the system support to the application program level (see Fig. 4):

1) *Distributed System and Communication Support Classes:* These are the classes which provide distributed system support.
2) *Transaction Primitive Classes:* These classes provide low level transaction primitives such as `unsafe commit`, and `cancel`.
3) *Atomic Transaction Classes:* Classes at this level use the primitive classes described in the previous level to materialize atomic units of work such as `actions` and `compensatable actions`.
4) *Scheduling Classes:* These implement the scheduling and synchronization processes described in Section 3.3.
5) *Application Program Class:* This class is used to implement user developed application programs (which could be the home loan application).

Dotted links between the individual classes represent inheritance relationships in the top-down direction, i.e., from the more general to the more specific class. For instance, the classes `Serial`, `Parallel`, etc. inherit the class `Scheduler`. Solid links represent a containment relationship. An example of that is the relationship between the class `Application_Prgm` and the different types of `Scheduler`s.

The low-level classes `Unsafe_Commit` and `Cancel` handle the unsafe-commit and cancel primitives. This should not be confused with component DBMS transaction primitives such as `commit` and `abort` are handled by component systems as already explained.

Atomic actions are supported by the two classes `Action` and `Compensating_Action`. These provide all types of functionality required for creating and communicating with atomic transactions running on local / remote database systems. Such actions are perceived by the native database system as another local transaction. To help ensure that actions are atomic, no two actions of the same intermediate activity can execute on the same database.

Schedulers are realized by four classes which inherit from the generic class `Scheduler` (see Fig. 4). This class is the most general scheduling class which provides properties inherited by all more specialized scheduling classes. Each of the remaining four scheduling classes is dedicated to one of the four types of schedulers mentioned in Section 3.3. All types of activities in the TOWE (except for flat transactions corresponding to actions) are instantiations of specialized scheduler classes.

In Table 1, we illustrate some of the most representative operations used in the classes depicted in Fig. 4. The table includes brief descriptions of the operations which are self-explanatory. Table 2 summarizes the PVM interface routines on which the library classes in Fig. 4 rely in order to perform interprocess communication and buffering activities.

## 4 Activity Specification and Programming

The TOWE model of computation supports decentralization by allowing loose coupling among the component databases: component databases need only have minimal knowledge (exported services) about each other. Developing a distributed application in TOWE corresponds to a high-level task, i.e., a long-running activity, which utilizes a number of data resources spread over a number of internetworked database systems. In the following, we present the programming of such a long running activity that implements the distributed workflow shown in Fig. 3. We assume that transaction programmers are familiar with exported data items and services of the component databases related to their applications. We will concern ourselves only with synchronization of actions belonging to the same activity; interactivity synchronization is handled as explained in Section 3.3.

To improve readability, we have simplified the definition of library classes by avoiding introducing syntactic constructs which lean heavily on Sather.

### 4.1 The Scheduler Class

Our view of concurrency is based on the unification of the concepts of class and process which leads to the notion of *process* object. Our model of concurrency employs inheritance for structuring and synchronizing objects: any instance inheriting directly or indirectly from any of the scheduling classes is a process object going through successive execution stages. All other objects are passive awaiting for a call to execute their methods. Concurrency can then be viewed as the concurrent execution of process objects and their interactions.

The synchronization classes encapsulate the state and behavior describing processes. The state encompasses the data structures needed for the communication, scheduling, and synchronization events that take place during the lifetime of a process object. After instantiation, a process object executes

its *work* routine which describes the process *script*, i.e., the sequence of actions it executes over its lifespan. The *work* statement materializes the behavioral part of a process object which provides the means to create other processes, actions, and objects at remote sites; and to request asynchronous execution of their features and to communicate with them. Fig. 5 illustrates the coding of the class *Transfer*, referred to as *transfer_money* in Fig. 3. This is a serial scheduler object that spawns two actions, each with its own independent thread of control, executed in the sequence specified in the *work* statement.

```
class TRANSFER{from_acct, to_account, sum} is SERIAL with
      const name: STR:= "transfer";
      balance: REAL;

      work is
            action.name("withdraw").arg(from_acct, sum);
            action.name("deposit").arg(to_acct, sum);
      end  (status); – –work
      – –block any consumer transactions which execute in parallel
            with this
      – –one until balance has been produced
      produced_values balance := set_value("deposit".balance);
end(status); – –transfer
```

Fig. 5. The *Transfer* process object.

The operations *name()* and *arg()* in Fig. 5 are used to supply the name and arguments of an action which may run at a local or remote site. The TOWE language supports location-independent process invocation: local and remote process calls are equivalent on the syntactic level, e.g., the actions *Withdraw* and *Deposit*. Since processes share memory, objects that appear as parameters of a communication between two process contexts, e.g., *sum*, *from_accn*, are passed by reference and not by copy.

The run-time system routes a client request via a proxy object to a target object that may reside in a remote address space. A *proxy* object is an object (similar to a client stub) that represents a server object on a client side, i.e., it runs on the same address space as the client [32].

A typical application may involve interactions with a variety of local and remote objects, but the proxy-objects make distributed-object interactions the same as local object interactions. To access a server object, the client simply performs a local invocation on the server's proxy. The proxy performs the actual remote invocation (cross-address space procedure call) and returns the result to the client. As proxy objects offer the same public interface as the server object, it is transparent to an application program whether it calls a proxy or a server (the actual implementation) object. Issues related to the client support infrastructure (such as proxy and surrogate objects) which enables the dispatching and activation of messages across machine boundaries can be found in [7].

Each data object is private and is accessible by only one process at a time. This guarantees that only one thread of control has access to it. Mutually exclusive access to shared data, i.e., the variable *balance* (*value dependency*) required for synchronization purposes, is transparently provided by the use of *produced_values* and *consumed_values* primitive constructs of the TOWE language class library. These constructs act like condition variables in that they block one or more processes by forcing them to wait until another has finished updating a shared data structure. This form of synchronization is known as *data-driven synchronization* and unifies data dependency with synchronization [3].

The correctness of the semantics of the transaction primitives used in TOWE, and, in particular, the *scheduler* classes, has been proven using the formal transaction specification framework ACTA. This formal framework supports specification of extended transaction models and reasons about transaction structure and correctness [5]. The formal semantics of the operations and correctness proofs can be found in [11].

### 4.2 Programming of Process Objects

The coding of the activity *Finance*, referred to as *Organize_Finance* in Fig. 3, presents some of the important features of the TOWE class library, see Fig. 6. Class *Finance* is a parallel scheduler as it spawns in its work statement actions that can run concurrently. These include an instance of the class *Transfer* as well as the conditional activity *More_Funds_Required*, see Fig. 3. This activity is conditional because the applicant may not need a personal loan. The user-defined class *Finance* may thus be defined as in Fig. 6.

```
class FINANCE{loan, property_value, loan_acct, applict_acct}
   is PARALLEL with
   const name: STR:= "finance";
   fail: BOOL:= FALSE;

   work is
      – – transfer loan from financial institution to applicant's acct
      action.name("transfer").arg(loan_acct, applict_acct, loan);
      – – need to borrow additional funds ?
      borrow:REAL := property_value - loan;

      if borrow > 0 then – – conditional transaction
            work is SERIAL-ALTERNATIVE with
              – – name of workpackage
              const work-name: STR:= "fund_alternative";
              – – hosts on which actions run
              hosts := "city_bank: ' personal_loan," "halifax:
                        home_equity_loan";
              action.name("personal_loan").arg(applict_acct, borrow);
              action.name("home_equity_loan").arg(applict_acct, borrow);
              status("fund_alternative") := commit_depend("transfer");
            end(status); – – work

      – – Commit dependency between actions
      – – personal loan, home_equity & money transfer needs to be
                  resolved
      if status("transfer") := COMMIT then
         status("fund_alternative") := COMMIT;
      else
         action is
           cancel("fund_alternative");
           fail := TRUE;
         end; – – cancel
      if (fail) then compensate("finance");
      else
         action.name("assess_applict").arg(applict_acct, loan, borrow);
   end; – –work
end;
```

Fig. 6. The *Finance* process object.

TABLE 1
SAMPLE LIBRARY CLASSES AND REPRESENTATIVE OPERATIONS

| Class | Routine Name | Brief Explanation |
|---|---|---|
| SCHEDULER (generic) | set_hosts() | set user-defined hosts for subtransactions. |
| | find_hosts() | find suitable hosts for subtransactions. |
| | spawn_sub() | spawn a subtransaction at a host. |
| | **name()** | specify name of a subtransaction. |
| | **arg()** | specify an argument of a subtransaction. |
| | is_a_sub() | if the argument is a subtransaction. |
| | decision() | make decision when scheduler is done. |
| | commit_protocol() | global commit protocol. |
| | get_group_size() | find out the size of a given group. |
| SCHEDULER (specific) e.g. SERIAL | exec() | execute subtransactions (one for each type). |
| | get_next_result() | receive result of the next subtransaction. |
| | get_all_results | receive result of all subtransactions (parallel). |
| | set_produced_args() | set the arguments for the next (serial) child. |
| | get_probe() | check for a particular message and receive it. |
| | abort_others() | abort the rest (choice schedulers). |
| DELEGATE | delegate() | accept and handle delegated transactions. |
| | service_assign() | assign a service to a host. |
| | is_host() | if the argument is an actual host. |
| | add_to_cluster() | add a host to a cluster. |
| | which_cluster() | to which cluster a host belongs. |
| | cluster_members() | return a list of cluster members. |
| | which_hosts() | hosts which provide a particular service. |
| ACTION | pass_values() | pass values to the consumers. |
| | get_values() | get values from the producer. |
| | pass_signal() | pass commit signal to the dependent tasks. |
| | get_signal() | get signal from the producer. |
| | decision() | make decision when the task is done. |
| CANCEL | main() | govern top-level task of the cancel process. |
| | extract_log() | extract info about top-level from cancel-log. |
| | form_comp_trans() | form the compensating transaction. |
| SATHER_PVM | pvm_routines | Sethar/PVM interface routines. |

TABLE 2
MOST IMPORTANT PVM INTERFACE ROUTINES

| Category | Routine Name | Brief Explanation |
|---|---|---|
| Virtual Machine | addhosts() | add one or more hosts to the virtual machine. |
| | delhosts() | delete one or more hosts from the virtual machine. |
| | config() | query about present configuration. |
| Process Control | mytid() | enroll in PVM and get a unique task id. |
| | spawn() | spawn a task at a host. |
| | exit() | exit from PVM. |
| | kill() | terminate a specified PVM process. |
| | parent() | find the process that spawned the present task. |
| Message Passing | mkbuf() | create a new message buffer. |
| | freebuf() | dispose a message buffer. |
| | initsend() | initiate the active buffer and specify encoding. |
| | pkint() | pack the active buffer with integer type. |
| | pkstr() | pack the active buffer with string type. |
| | upkint() | unpack the active buffer with integer type. |
| | upkstr() | unpack the active buffer with string type. |
| | send() | non_blocking send. |
| | sendsig() | send a Unix signal to a PVM process. |
| | mcast() | non_blocking multicast. |
| | recv() | blocking receive. |
| | nrecv() | non_blocking receive. |
| Dynamic Groups | joingroup() | join a group (define if non-existent). |
| | lvgroup() | leave a group. |
| | gsize() | query about group size. |
| | bcast() | broadcast to all group members. |
| Synchro-nization | barrier() | increment counter and wait for others to call. |
| | bufinfo() | find information about specified buffer. |
| | probe() | check if a msg from a task with a tag has arrived. |
| | pstat() | find status of a specified PVM process. |

Class *Finance* introduces a serial alternative (see *Fund_Alternative* in Fig. 3) scheduler nested within a parallel scheduler (see *Finance* in Fig. 6). The serial alternative scheduler semantics specifies that either of the two activities is to succeed (in the specified sequence) for this scheduler to succeed. Notice that one may specify the sites on which the activities *Personal_Loan* and *Home_Equity_Loan* may run. The variable *hosts* is set to the hosts where the transaction ran. There is also a commit dependency between the activity *Fund_Alternative* and *Transfer*. In other words, the s*tatus* of the activity *Fund_Alternative* is forced to take the status of *Transfer*, i.e., they both either succeed or fail. Failure means that activity *Finance* must also fail.

Schedulers of any type generally have multiple children. In some cases, it is useful to have a single child tried at a number of sites, i.e., we may have *identical actions* executed concurrently at several sites. This case effectively results into multiple identical actions running on different hosts to accelerate concurrency. For example, consider the activity *Find_Credit_Union* in Fig. 7. This scheduler reflects the situation where an applicant is interested in obtaining support from any credit union, tries several, and accepts the first offer that is received. The parallel alternative scheduler requires only one transaction to succeed in order to commit successfully.

```
class FIND_CREDIT_UNION {to_acct, amount} is
                    PARALLEL_ALTERNATIVE with
      const name: STR:= "find_credit_union";
      work is
            hosts := some.domain – –customized
            action.name("support_home_loan").arg...
      end; – –work
end;
```

Fig. 7. The *Find_Credit_Union* process object.

In the TOWE language, actions may get two types of arguments. The first type of argument is the one that is provided directly by the programmer as already shown in the previous examples. The other type of argument is the one provided indirectly by other actions of the same parent. For example, an action may open an account and then pass the entire account object to another action to operate upon. The TOWE class library provides a simple mechanism for this case. The "work" routine of the client object exports the objects in question packaged in a string. Consider the client objects *Insure_Property* and *Update_Title* specified in the object list of the object *Open_Home_Loan_Account*, see Fig. 8. Each set of arguments must be attached to the name of the client action by a ":". As soon as these arguments are received by the client objects they unblock. Multiple sets are separated by a semicolon. For instance, consider the process *Open_Home_Loan_Account* in Fig. 8. This is a more explicit way for processes to communicate. Notice that the object passed from the server object *Open_Home_Loan_Account* to the client objects *Insure_Property* and *Update_Title* is of type *Account* which is not an active but rather a conventional data object.

```
class OPEN_HOME_LOAN_ACCOUNT{applict_info} is ACTION
                    with
      const name: STR := "open_home_loan_account";
      home_loan_account: ACCOUNT;
      work is
      ...
      res := "home_loan_account':insure_property; home_loan_account:
                    update_records";
      end;
end;
```

Fig. 8. The *Open_Home_Loan_Account* process object.

### 4.3 Visibility of Intermediate Results

As actions normally communicate via shared objects and data structures. Once an activity commits, its effects become automatically visible to other activities. To avoid corrupting shared data structures and violating internal consistency, we have introduced the notion of *unsafe commit* for such cases.

Each intermediate activity in the TOWE may commit atomically once its internal actions have executed successfully. This has the effect of making an action's results visible to other concurrently executing activities. This means that if an activity aborts, only the action interrupted by a failure can be rolled back unilaterally. Atomicity may be achieved by semantically undoing other actions that executed before by issuing compensation transactions. A better alternative around this problem is the use of unsafe commits. Committing an action unsafely, in the TOWE, comprises two basic parts:

1) Logging sufficient information for the possible cancel process.
2) Committing the action and releasing the locks it holds.

The second part is the same as the *commit* operation which is implemented by any of the component database systems in the distributed client/server network. The class *Unsafe_Commit* and *Cancel* are concerned with managing the cancel-log. A single compensating transaction is formed, executed, and committed for all actions of a failed intermediate activity. This is a specialized multilevel compensating transaction which is constructed automatically by the system and relies heavily on information stored on the cancel-log. This approach achieves superior results when compared with simple compensation transactions which are normally user created, operate on a single level, and have to be explicitly supplied with the original values of the updated data items.

The cancel-log contains sufficient information to specify where to backtrack if data items are corrupted. It contains the names, action identifiers, and types of actions, information about their hosts on which they execute as well as information about the internal composition of activities, i.e., their descendants. The two main operations of the class *Cancel*, *extract_log()*, and *form_comp_trans()* extract information from the cancel-log and form a tree of compensating transactions belonging to any action to undo its entire effects. This could be, for example, the case if the parallel scheduler process *Organize_Finance* fails.

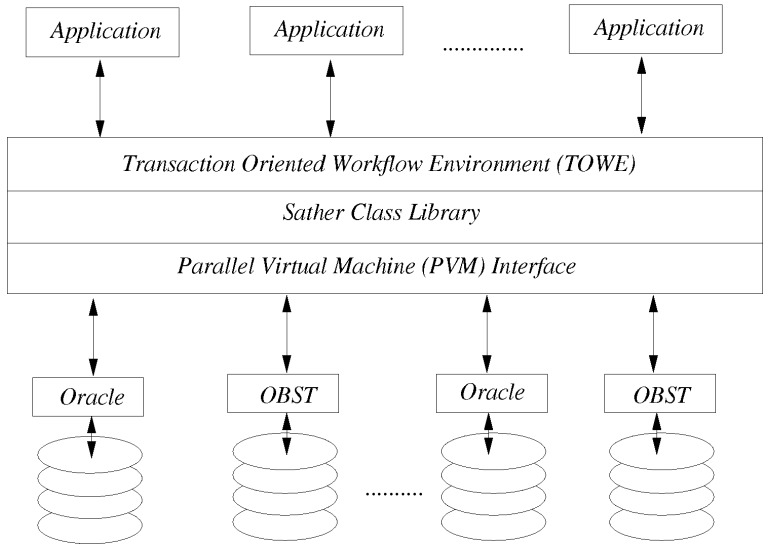



Fig. 9. The TOWE system architecture.

Abort recovery follows a simple approach along the lines of Argus [18]. A data item has associated with it a stack of versions. When an action updates a data item, it copies the current version of this data item from the top of the stack, and pushes the new onto the stack. Only subsequent updates are done on the top stack item. When an action commits, its version (if any) at the top of the stack replaces its parent's version. When an action aborts, its version (if any) at the top of the stack gets discarded. All changes to local/remote databases are then undone by running compensating transactions in a similar way as previously explained.

## 5 Implementation and Experience

The TOWE system has been implemented on a distributed Unix platform. The collection of the host machines included various types of Sun machines (Sun-IPX, Sun-LX, and Sun5) running Solaris 2.4. They are connected in a local area network using NFS. A large part of the code was written in Sather—it consisted of approximately 5,000 lines of code. The current prototype makes use of two off-the-shelf database management systems: the *Oracle* database system and the object-oriented research prototype OBST.

Interaction of the TOWE with *Oracle* is facilitated through a programmatic interface. In particular, *Oracle*'s API has a significant advantage as it allows ad-hoc SQL queries in a host language. For instance, the *Pro*C* tool [26] can convert a program written in C with SQL statements into a C program that can access and manipulate data maintained by the DBMS. Due to the fact that Sather source programs are compiled into portable C code, they can be easily linked to the TOWE script language. In addition, native transactions can be linked with the PVM library for communication and synchronization purposes and they can be called in the user-defined classes. At run-time, the format of the transactions is actually not different from the objects format written in Sather because Sather programs are compiled into C.

The other database product we used in our implementation is the object-oriented database OBST. It is a public domain object-oriented persistent storage system implemented as an enhancement to existing programming languages [4]. The OBST data model provides mechanisms for defining types in terms of modules called schemas. The basic system configuration includes a schema compiler, a library of predefined classes, a graphical object browser, and a structuring tool. OBST also offers a mechanism to incrementally load methods. This feature enables programs to deal with objects whose type is defined dynamically (at runtime). OBST supports the execution of atomic transactions written in *C++*. The OBST system also supports the linking of transactions with the *PVM* class library. Transactions are separately compiled and subsequently linked with objects stored through the TOWE. The dynamic group mechanism for synchronization was possible because both languages, Sather and *C++*, interface almost seamlessly with the *PVM* system calls.

Our experiments have consisted of running applications to test the core features of the TOWE system. The TOWE scripts provided the means of interaction with the system. The PVM library offered transparent communication among the different participating sites. A high level description of the current TOWE implementation is shown in Fig. 9. The running example used in this paper, the *Banking Loan Workflow Application*, was implemented and successfully tested. The required data repositories were simulated using the aforementioned database systems. In the context of our experiments, six such database servers were deployed. Half of them simulated regular banking corporations and the remaining were used to represent the participating credit unions. Applications of customers seeking loans were submitted from client workstations. TOWE application scripts then delegated the various tasks to the database servers for processing. PVM calls enable applications to execute on the necessary network nodes. Once intermediate results were collected, the TOWE system directed the remaining tasks to be executed as in the application script.

In our application, all financial institutions maintained profiles for various categories of customers. They made decisions on applications and grant loans based on various

loan applicant profiles. The response times obtained for the workflow representing the *Banking Loan Application* consistently ranged from 30.851 seconds to 63.861 seconds depending on the number of the databases involved and the network traffic. Times were accrued by the clients executing the workflow scripts.

## 6 Concluding Remarks and Future Work

In this paper, we presented a Transaction Programming Environment (TOWE) based on class libraries which are realized on a unification of concepts from object-oriented programming with distributed computing and open-nested transaction facilities. We illustrated the use of TOWE by means of a distributed workflow application modeling a home loan planning activity.

We prototyped the TOWE on a concurrent object-oriented framework that combines advanced object-oriented facilities with user interface primitives for the invocation of processes, broadcasting, synchronization via barriers, mutual exclusion, and exception handling. The TOWE allows for more flexible concurrency control mechanisms by offering programming constructs such as various types of scheduling dependencies among processes, blocking/non-blocking synchronization primitives, safe and unsafe commitment primitives, compensation, and contingency transactions.

Our experiences with TOWE have revealed the short comings of using a static type of inheritance in connection with synchronization. More flexible mechanisms are needed for dynamic inheritance (for overriding the behavior of parent classes at run-time) and for providing flexible combinations of inherited functionality in a single module. Techniques used in reflective programming languages and systems [1], [20] may in fact provide a more appealing solution.

Our future work will concentrate on extending the functionality of the current TOWE implementation. There are three important areas that we target:

1) support for a variety of database systems
2) support for modeling organizational aspects which do not require transactional properties, and
3) user interface support.

Our experiments have so far consisted of running banking applications to test the salient features of TOWE on two heterogeneous database systems: (the object-oriented database system) OBST and ORACLE. In the future, we plan to construct other types of applications involving few other commercial and research database systems.

In the second case, an important requirement is the ability to define and influence the execution of workflow processes according to the characteristics and policies of an organization. Hence, we are currently extending the concept of *roles* that we have developed in [29] and [28] for workflow systems so that when applications are developed, it is possible to specify responsibility for the execution of activities. This can be accomplished in terms of roles where all persons that undertake this role are eligible to execute a certain activity. For example, within an organization, people may have several roles such as manager, programmer, designer, business analyst, etc., and a role can be assigned to many persons. The role mechanism provides a great deal of flexibility when executing processes within a workflow environment.

On the user interface front, activities concentrate on developing a *process definition tool* along the lines proposed by the *Reference Model* of the *Workflow Management Coalition (WfMC)*, an international forum striving to standardize workflow management products [14]. This interface will ultimately map workflow process specifications relying on a common interchange format to workflow execution services. Such workflow process specifications may include process start and termination conditions; identification of inter and intraprocess activities (including identification of data types and access paths); definition of flow rules; and information about resource allocation.

Our efforts will also include experimentation with WWW technology as a general user interface paradigm for workflow applications developed in TOWE. This requires appropriate mappings of the TOWE processes to a set of cooperating CGI (Common Gateway Interface) scripts. With the help of such scripts, one can process workflow information received through HTML forms. CGI scripts could easily interact with TOWE applications and existing Oracle or OBST databases.

## Acknowledgments

We wish to thank the reviewers of this article for their many insightful and constructive comments. Part of this work was supported by a Large Australian Research Council (ARC) grant number 95–7–191650010. A. Delis was partially supported by the Center for Advanced Technology in Telecommunications (CATT) in Brooklyn, New York.

## References

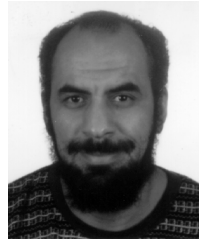
[1] G.A. Agha, "Concurrent Object-Oriented Programming," *Comm. ACM*, vol. 33, no. 9, Sept. 1990.
[2] A. Beeharry, A. Bouguettaya, and A. Delis, "On Distributed Persistent Objects for Ineroperable Data Stores," *Information Sciences, An Int'l J.*, vol. 91, pp. 1-32, May 1996.
[3] D. Caromel, "Toward a Method of Object-Oriented Concurrent Programming," *Comm ACM*, vol. 36, no. 9, Sept. 1993.
[4] E. Casais, M. Ranft, B. Schiefer, D. Theobald, and W. Zimmer, "OBST—An Overview," Technical Report FZI/039.1, Forschungszentrum Informatik (FZI), Karlsruhe, Germany, June 1992.
[5] P. Chrysanthis and K. Ramammritham, "Acta: The SAGA Continues," *Transaction Models for Advanced Database Applications*, A. Elmagarmid, ed. San Mateo, Calif.: Morgan Kaufmann, Feb. 1992.
[6] J.F. Colin and J.M. Geib, "Eiffel Classes for Concurrent Programming," *Proc. 1991 TOOLS Conf.* Prentice Hall, Jan. 1991.
[7] D. Edmond, M.P. Papazoglou, and Z. Tari, "ROK: A Reflective Model for Distributed Object Management," *RIDE-DOM: Research Issues in Data Eng.*, M. Tamer Özsu, O. Bukhres, and M.-C. Shan, eds., Taiwan, Mar. 1995.
[8] C. Ellis and G. Nutt, "Modeling and Enactment of Workflow Systems," *Application and Theory of Petri-Nets*, Lecture Notes in Computer Science vol. 691. Springer-Verlag, 1993.
[9] J. Eppinger, L. Mummert, and A. Spector, *Camelot and Avalon: A Distributed Transaction Facility*. San Mateo, Calif.: Morgan Kaufmann, 1991.
[10] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, "PVM 3 User's Guide and Reference Manual," Technical Report ORNL/TM-12187, Oak Ridge Nat'l Laboratory, Sept. 1994.

[11] M.S. Haghjoo, "Transactional Actors in Cooperative Information Systems," PhD thesis, Dept. of Computer Science, Australian Nat'l Univ., Canberra, Australia, 1995.
[12] M.S. Haghjoo and M.P. Papazoglou, "Tractors: A Transactional Actor System for Distributed Query Processing," *Proc. 12th Int'l Conf. Distributed Computing Systems*, Yokohama, Japan, June 1992.
[13] N. Haines, D. Kindred, G. Morrisett, S. Nettles, and J. Wing, "Composing First-Class Tranasactions," *ACM Trans. Programming Languages and Systems*, vol. 16, no. 6, Nov. 1994.
[14] D. Hollinsworth, "The Workflow Reference Model," Technical Report TC00-1003, Workflow Management Coalition, Brussels, Nov. 1994. `Http://www.aiai.ed.ac.uk/WfMC/DOCS/refmodel/rmv1-16.html`.
[15] M. Hsu, "Workflow Systems," *IEEE Data Eng. Bulletin*, vol. 18, no. 1, Mar. 1995.
[16] P. Hung, H. Yeung, and K. Karlapalem, "Capbased-ams: A Capability-Based and Event-Driven Activity Management System," *Proc. 1996 SIGMOD Conf.*, Montreal, June 1996.
[17] N. Krisnakumar and A. Sheth, "Managing Heterogeneous Multi-System Tasks to Support Enterprise-Wide Operations," *Distributed and Parallel Databases*, vol. 3, no. 2, June 1995.
[18] B. Liskov, "Distributed Programming in ARGUS," *Comm. ACM*, vol. 31, no. 3, Mar. 1988.
[19] C. Martens and F. Lochovsky, "OASIS: A Programming Environment for Implementing Distributed Organisational Support Systems," *SIGOIS Bulletin*, vol. 12, nos. 2-3, 1991.
[20] H. Mashuhara, S. Marsuoka, T. Watanabe, and A. Yonezawa, "Object-Oriented Concurrent Reflective Languages Can Be Implemented Efficiently," *Proc. 1992 OOPSLA Conf.*, ACM, Sept. 1992.
[21] R. Medina-Mora, T. Winograd, and R. Flores, "The Action Workflow Approach to Workflow Management Technology," *Proc. 1992 Conf. Computer-Supported Cooperative Work*, Toronto, Nov. 1992.
[22] B. Meyer, "Systematic Concurrent Object-Oriented Programming," *Comm. ACM*, vol. 36, no. 9, Sept. 1993.
[23] J.E.B. Moss, *Nested Transactions: An Approach to Reliable Distributed Computing*. Cambridge, Mass.: MIT Press, 1985.
[24] M. Nodine, N. Nakos, and S.B. Zdonik, "Specifying Flexible Tasks in a Multidatabase," *Proc. Second Int'l Conf. Cooperative Information Systems*, Toronto, May 1994.
[25] S. Omohundro and C. Lim, "The Sather Language and Libraries," Technical Report TR-92-017, Int'l Computer Science Inst., Berkeley, Calif., Mar. 1992.
[26] *Oracle, The Relational Database Management System*, Pro*C User's Guide, version 1.1. Technical report, Oracle Corp., Redwood Shores, Calif., 1992.
[27] M.P. Papazoglou and N. Russell, "A Semantic Meta-Modeling Approach to Schema Transformation," *Proc. Fourth Int'l Conf. Information and Knowledge Management*, Baltimore, Dec. 1995.
[28] M.P. Papazoglou and B. Krämer, "Representing Transient Object Behavior," to appear in *VLDB J.*, 1997.
[29] M.P. Papazoglou, B. Krämer, and A. Bouguettaya, "On the Representation of Objects with Polymorphic Shape and Behavior," *Proc. 13th Int'l Conf. Entity-Relationship Approach*, Manchester, Dec. 1994.
[30] M. Rusinkiewicz, P. Krychniak, and A. Cichocki, "Towards a Model for Multidatabase Transactions," *Int'l J. Intelligent and Cooperative Information Systems*, vol. 1, no. 3, Sept. 1992.
[31] H. Schuster, S. Jablonski, P. Heinl, and C. Bussler, "A General Framework for the Execution of Heterogeneous Programs in Workflow Management Systems," *Proc. 1996 Int'l Conf. Cooperative Information Systems*, Brussels, May 1996.
[32] M. Shapiro, "Structure and Encapsulation in Distributed Systems: The Proxy Principle," *Proc. Sixth Int'l Conf. Distributed Computing Systems*, May 1986.
[33] W. van der Aaalst, "Petri-Net Based Workflow Management Software," *Proc. NSF Workshop: Workflow and Process Automation in Information Systems*, Athens, Ga., May 1996.
[34] H. Waechter and A. Reuter, "The Contract Model," *Transaction Models for Advanced Database Applications*. San Mateo, Calif.: Morgan Kaufmann, Feb. 1992.
[35] J. Widom and S. Ceri, *Active Database Systems: Triggers and Rules for Advanced Database Processing*. San Mateo, Calif.: Morgan Kaufmann, 1996.
[36] D. Wodtke, J. Weissenfels, G. Wikum, and A. Kotz Dittrich, "The Mentor Project: Steps towards Enterprise-Wide Workflow Management," *Proc. 12th IEEE Int'l Conf. Data Eng.*, New Orleans, Feb. 1996.
[37] C. Woo and F. Lochovsky, "Supporting a Distributed Office Problem Solving in Organizations," *ACM Trans. Office Information Systems*, vol. 4, no. 3, Mar. 1986.

**Mike Papazoglou** is a full professor and director of the INFOLAB at the University of Tilburg, The Netherlands. Prior to this, he was a professor and head of the School of Information Systems at the Queensland University of Technology in Australia.

Dr. Papazoglou is the founding editor of the *International Journal of Intelligent and Cooperative Information Systems* (World Scientific) and a member of the editorial advisory board of six international information and technology journals and a reviewer for several journals and conferences. He has authored and edited more than 80 books, journal articles, and refereed conference papers. He is a Golden Core member of the IEEE and a distinguished visiting lecturer for IEEE in the Asia/Pacific region. He has served as general and program chair for IEEE sponsored conferences and has been a program committee member for numerous conferences.

**Alex Delis** received his PhD and MSc in computer science from the University of Maryland at College Park in 1989 and 1993, respectively, and his BS in computer engineering from the University of Patras, Greece, in 1985. He is currently an assistant professor of computer science in the Department of Computer and Information Science at the Polytechnic University in Brooklyn, New York. His research interests are in databases, information systems, and software engineering. Dr. Delis was the recipient of the Outstanding Paper Award at the 14th IEEE International Conference on Distributed Computing Systems (ICDCS) in 1994. He is a member of the IEEE, ACM, and the New York Academy of Sciences.



**Athman Bouguettaya** received his MSc and PhD in computer science from the University of Colorado at Boulder in 1987 and 1992, respectively. He joined the School of Information Systems at Queensland University of Technology, Brisbane, Australia, in 1993. He has conducted research and published in several areas including database interoperability, object-oriented databases, database clustering, and workflow management systems.

**Mostafa Haghjoo** received his PhD and MSc in computer science from the Australian National University and George Washington University, respectively. He has worked as a tutor at the Australian National University and as an associate lecturer at the Queensland University of Technology. He is currently a lecturer in computer science at the Iran University of Science and Technology in Tehran. His research interests are in databases, distributed programming, and object-oriented systems.