General Purpose Work Flow Languages

Alexander Forst, eva Kühn and Omran Bukhres

In: International Journal on Parallel and Distributed Databases, special issue on *Software Support for Work Flow Management*, Kluwer Adademic Publishers,
Vol. 3, No. 2, April 1995.

Journal, 0, 1–33 (0)

General Purpose Work Flow Languages*

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

OMRAN BUKHRES bukhres@cs.purdue.edu
Indiana Center for Database Systems, Purdue University, Department of Computer Sciences, West Lafayette, IN 47907, USA

Abstract. Work flow management requires language support for work flow specification and task specification. Many approaches and systems for work flow management therefore offer at least one new language for work flow specification; task specification is usually done in a traditional language. This is motivated in particular by the fact that many components already exist and the task of the work flow tool is the specification of the interaction between these components. The intention of this article is to demonstrate that a general purpose programming language can serve both aspects. We do not really see the need to develop yet another language that a user or application programmer must learn. If an existing programming language like C or Prolog is extended towards work flow capabilities, it is easy to reuse autonomous existing software components and to build interfaces among them.

Keywords: Work Flow Language, Coordination System, Heterogeneous Transaction Processing

1. Introduction

Work flow systems have their origin in the office automation efforts since the 1970s. Today, work flow management comprises the description of units of work, and the specification of data and control flow among the many different activities and interactive tasks, including constraints and dependencies. The modeling of activities within a computer system is a necessary condition to provide software support for work flow management. This support is provided through advanced mechanisms and approaches for the supervision of activities in an organization that are acceptable to the users and can be integrated into the existing structures. When a user interacts with a software system in a work flow environment, a tool must provide the necessary programming support for cooperative work with other users, that possibly use different software systems such as database systems, e-mail, CASE tools, text editors, spread sheets, and other office-automation software. Many enterprises use multiple information processing systems that, in most cases, were developed separately to automate different functions. These systems are often managed independently, yet contain related and overlapping data.

* The work is supported by the Austrian FWF (Fonds zur Förderung der wissenschaftlichen Forschung), project "Multidatabase Transaction Processing", contract number P9020-PHY in cooperation with the NSF (National Science Foundation).

The central issues to work flow management systems are: the combination of these distributed, autonomous systems, applications, and data repositories, programming support for cooperative work, and reliable and transparent communication. Communication serves not only to pass intermediate results and information, but also to control the distributed execution of the work. Open architectures and advanced transaction models are important prerequisites for the coordination and the verification of correct execution of these tasks and interactions.

The rapid growth of advanced applications involving distributed transaction processing has resulted in the development of many distributed models and languages. With today's development of sophisticated and complex distributed applications, software support providing a more advanced model than the embedding of message passing into a sequential language is needed. Instead of developing a new work flow language from scratch, we propose to extend existing, general purpose programming languages by a new coordination toolkit and equip them this way with work flow capabilities.

This article is structured as follows: in the rest of this section we analyze necessary features of a work flow language, mention related technologies to work flow management, and explain our toolkit approach. In Section 2, we summarize the work flow language C&Co which is an extension of C by the coordination toolkit. In Section 3 we compare C&Co with other general purpose languages that offer work flow facilities. In Section 4 we present a solution for a work flow problem in terms of C&Co. In Section 5 we discuss C&Co as a work flow management tool and point out its advantages for important classes of work flows.

1.1. Work Flow Language Requirements

The tasks in a work flow environment can be specified in a traditional, sequential programming language. These units might be pre-existent software modules, autonomous services in a network, or data repositories that shall be included into the entire work flow. The specification of the work flow declares the interaction patters between the single units and thus requires more advanced linguistic support. We will refer to a language that manages task and work flow specification with the notion *work flow language*.

1.1.1. Advanced Control Mechanisms

The explicit modeling of the control flow between tasks needs constructs to specify execution state dependencies and data dependencies. The execution of a task can depend on the progress (success or failure) of another task, or on the results that the other task has produced. It must be possible to check the execution states of all concerned tasks and to constrain the control flow depending on these values. This will effect the degree of the allowed concurrency in the work flow.

Throughout the article we will use an example that shows a work flow for the arrangement of a business trip for a university member, including the negotiation for adequate fundings and the flight reservation. In terms of this example, the flight reservation depends on the success of the activity that organizes a travel fund. This includes an execution state dependency (the flight booking must not be done while the other activity is still running), and a data dependency (the result must be a valid fund).

A further important issue is the specification of sequential and parallel execution. A global task may consist of several activities which run on distributed sites. We need language constructs to specify when the global task is accomplished. This can be the case if all activities have completed, or if a well-defined subset of them has terminated successfully, e.g., if exactly one activity has completed, if k of n activities were successful, or an even more complex statement specifying an arbitrary pattern of the activities. Semantically, if a subset of activities suffices, we speak of *alternatives*. More specifically, the task at hand can be fulfilled in several ways which leads to non-deterministic specifications, if all activities are tried in parallel. In Section 1.1.4 below, we will explain the impacts of specifying alternatives on transaction management.

More advanced control mechanisms require the specification of external constraints, (e.g., time dependencies, dependencies on external resources), retry conditions, and interrupts. Some examples: if the travel cannot be arranged before the conference starts that the professor wants to visit, the arrangement activity need not proceed. If the funds are granted, but the flight cannot be booked, the flight booking (possibly at another airline database) shall be retried, until it is successful or the last possible departure date is exceeded. If the professor falls ill, the task shall be interruptable.

1.1.2. Support for System Integration

A work flow language is responsible for the control of the communication between the single activities. As we assume that some software modules already exist and cannot be changed, the work flow language has to communicate with them via their usual interfaces. The work flow tool may run on several sites and thus must provide a reliable communication facility that is resistant against network and system failures. The communication with an existing software system can be handled by having one thread implemented in the work flow language running on the same site where the autonomous software system is located so that local communication facilities like pipes and files can be used. The remote (and more complicated) communication via the network can then rely on the communication facilities of the work flow language. The work flow language can be seen as the glue between the communicating pieces.

For example, the work flow language must reliably transport the request of the professor for a travel grant to the head of the department. The request must arrive there exactly once. The answer must be transported back to the professor's office in a reliable way. If it is lost, the fund will be reserved, but the professor is not informed about it and thus cannot visit the conference. If the grant approval arrives several times, it might happen that the professor thinks that he has received several funds and decides to spend too much money for the travel.

If the integration of software modules is supported, the work flow language can use such a module as a persistent object store. It is not necessary to implement all features in the work flow language that can be imported ("bought") from other servers in the network. The basic idea should be to re-use existing software and services. The main task of a work flow language is to organize and coordinate.

User interaction is essential, if work flow management has to cope with unexpected situations that cannot be predicted at design time. A work flow system may run into situations, where a user must provide a decision. For example, the travel agency might ask the professor, if he is willing to take an expensive ticket that does not fulfill the original price constraint, because it is the last ticket on this flight.

User interaction can also lead to ad-hoc specified work flows, which for example are used only once and therefore not statically stored. A further advanced possibility that a user might expect from his work flow tool is to change an already running specification dynamically.

1.1.3. Object Model

Work flow specifications semantically represent complex tasks that can be decomposed into subtasks. Each task can be seen as a contract between the institution, where the software system runs that is used to accomplish the task, and the client issuing this task via his work flow tool. The provider of a task must clearly specify its interface and the effects it has. The client need not be concerned about the implementation of the task.

Interface specification is supported at best by using the object model [42]. Objects encapsulate data and behavior and offer a well-defined interface via their methods. Objects shall be the units that can be shared among concurrent activities. In common object models, each method invocation is done atomically: the object behaves like a monitor where exclusive access is guaranteed to the clients.

More advanced concurrent and distributed object models provide transactions where several methods on objects can be performed atomically. Actions may be nested: the invocation of one method may trigger other methods on other objects.

We will later show how the coordination model that we propose supports objects. The coexistence of objects and concurrency in our model differs from traditional ones in that a single method can access more than one object at the same time within a transaction. Objects in our model are persistent, write-once entities that can be shared by parallel processes and thus serve for communication. They do not replace object stores, but the coordination model supports interfaces to existing data repositories (like very large databases) and uses objects to communicate these data between different sites.

Software systems can be more advanced systems than simple file stores: they can provide a data model so that their integration requires advanced data representation capabilities. Object-orientation is the methodology that supports semantic data integration [10]. An object model is a prerequisite for providing object-orientation.

1.1.4. Transaction Capabilities

In the above, we have used the notions "task" and "activity" which we would like to replace by "transaction". A classical transaction guarantees the atomicity, consistency, isolation, and durability of the task execution (ACID property [7]). Unfortunately, the ACID properties are too strict for heterogeneous environments. The specification of alternative transactions requires a model where the atomicity requirement cannot be fulfilled, e.g., a flight booking transaction is fulfilled, if one of several reservations at different airline databases succeeds. Extensions to specify such situations are needed. As another example, long-living or possibly continuous activities are intractable under the isolation property. The isolation property is also an obstacle, if the transaction models cooperative work, where subtransactions have to exchange results, before the global transaction completes.

For these reasons, several advanced transaction models have been recently proposed that relax these ACID properties in some way [22]. For example, the Flex Transaction model [21], [36] was designed especially to meet the requirements of heterogeneous environments and for multidatabase systems. It provides *function replication* and *semantic compensation* of nested transactions by relaxing the isolation property of classical transactions.

Function replication allows the specification of alternatives and is needed to substitute a failing service by another one that fulfills a task in an equivalent way (partial failure recovery).

Relaxation of the isolation property is important in coordinating local database systems. A subactivity at a local database system will acquire locks and keep them until the global activity terminates. Therefore, subtransactions are allowed to complete (in the database sense: commit) before the global task is finished (see also [25], [22]). This makes long-lasting cooperative and communicating transactions possible. Semantic compensation is motivated by the relaxation of the isolation property. A completed action might not be needed because of function replication or might be aborted if the global task fails. Here, a successfully completed transaction must not be rolled back because other autonomous processes might already have seen its effects and might have based their computations on them. Thus, if a process decides subsequently that its computations are no longer valid, it must *semantically compensate* for them. This means that a user-defined compensate action (procedure, function) is automatically activated. In summary, only actions that are compensatable may complete early.

The necessary extensions for a work flow language to provide the features of the Flex Transaction model include:

Transactions. The start and completion (commitment) of a transaction must be explicitly controllable.

Write and Read of Shared Data. Write operations must be collected until the transaction on the objects is committed so that all the values of the data objects become visible in an atomic step. If a datum cannot be written, the transaction must abort.

Compensate Actions. The run time selection of compensations, i.e., procedures that are activated if the global transaction aborts, to compensate, i.e., semantically undo, completed transactions shall be possible within transactions.

Prepared Phases. If it is sure that a transaction can commit, i.e., all assignments of values to the objects are possible, a procedure is executed that shall be specifyable by the programmer. This mechanism can be used to signal concurrent processes that this very transaction is committing. In database terminology this is called the prepared phase.

In the Flex Transaction model, transactions are either *compensatable* or *non-compensatable*. Compensatable transactions commit immediately. Compensate actions must be associated with compensatable transactions to allow rollback of global transactions. The systems on which non-compensatable transactions execute, must support a two-phase commit protocol so that their commitment can be delegated to the global transaction. Thus, the granularity of commitment and of compensation can be controlled explicitly.

1.2. Related Technologies

The objectives of *multidatabase systems* [9], [40] are related to those on work flow management. A multidatabase system aims at the integration of autonomous software systems (*legacy systems*). It can either provide a very tight coupling, hiding all differences of the local systems from the user, or provide a loose integration leading to more flexibility. A multidatabase system with loose integration resembles a programming tool that supports the management of distributed activities.

A multidatabase system originates from the need to integrate local database systems. This definition has been expanded towards the integration of arbitrary software systems. A multidatabase system is highly data oriented and aims at a uniform, global integration. The interaction of users should be avoided, if possible.

In contrast, the programming aspect in work flow management systems is much more pronounced. As stated above, tasks and dependencies must be specified. Frequent changes must be accounted for. The formulation of ad-hoc work flows should also be possible. Work flow management started with the integration of local services of one institution (like mailers, text systems, etc.) and it expands towards the integration of services of several autonomous institutions.

In summary, a multidatabase system has to cope with semantic data integration and heterogeneous transaction processing. Because work flow systems are highly dynamic, semantic data integration is less important here. However, transactions are of equal importance as in multidatabase systems.

1.3. Toolkit Approach

The features of advanced transactions can either be supported by a specialized operating system layer or by a distributed and parallel programming language that provides concurrency, communication, and synchronization. The language must maintain persistent objects and have a well-defined failure behavior. Traditionally, the operating system provides libraries to support the programmer. However, sockets, RPC, and related communication techniques are not adequate for advanced transaction specifications. A more advanced language support is needed.

Recently, specialized languages like IPL (InterBase Programming Language [16]) have been developed to implement Flex Transactions. However, many software components are written in commonly used programming languages like C. Thus, our objective is to extend existing languages with coordination properties, that are powerful enough to provide the features of the Flex Transaction model. The result is a language-independent coordination framework, called the *coordination kernel* [31] that can either be called in form of library functions, or be embedded smoothly into existing programming languages. We term a language (or system) $\mathcal{L}$ which is extended by this coordination toolkit "$\mathcal{L}$&Co" ("$\mathcal{L}$ plus coordination"). Every &Co system can be called from another &Co system or directly from the shell of the kernel. The extension of Prolog, also called $^{\mathrm{V}}\mathrm{P}_{\mathrm{L}}$ (Vienna Parallel Logic [34]), is called "Prolog&Co". The extension of C, originally proposed in [30], is called "C&Co".

C&Co permits simultaneous execution of tasks, thus achieving a higher degree of parallelism in the work flow environment. Compensation and function replication are builtin properties of the language. Programmers and users can specify dependencies among tasks and subtasks and the execution order of a global activity. The dependency specifications support functionally equivalent alternatives for the goals of transactions. This flexibility increases the tolerance towards failures of the individual subtasks.

C&Co supports the integration of programming languages, database features, and operating system features and the use of preexisting software systems for a wide range of applications. The design goal of C&Co was the attainment of a high-level of abstraction of the architecture through a conceptually high model for communication based on advanced transactions on shared objects.

2. The Coordination Language C&Co

This section describes how C&Co extends C's semantics without destroying the typical C programming style. We introduce into C the concept of communication variables to enforce persistency and extend the semantics of C's operators to deal with them. A communication variable is just like any other normal variable in C except that it can be shared between parallel and distributed processes. The processes always have the same view of the communication variable. A communication variable is either *undefined* or it can be assigned a value once (rigorous single assignment property). Thus, every process which has access to this communication variable can rely on its value because it will never change in the future.

Values are assigned to communication variables only within transactions (see Section 2.3). More precisely, the value of a communication variable becomes visible to other processes only after the writing transaction commits. In addition, communication variables survive system failures and provide a reliable communication medium. The implementation of communication variables in a distributed environment is described in [31], [35].

2.1. Creation of Communication Variables

Communication variables are shared between several processes and are defined with the `comm` type qualifier. It can prefix any C data type declaration, causing the creation of a communication variable of the specified type. The advantage of this approach is that all C types remain, but variables can be defined to be communication variables which causes them to exist in a globally shared space.

Program 1 (*Static Creation of Communication Variables*)

```
comm int i_object;       /* creation of integer communication variable */
struct date
    {
    int dd, mm, yy;
    };
comm struct date date_object;  /* communication variable of type date */
struct node
    {
    int key;
    comm struct node *left, *right;
    };
comm struct node *distributed_tree;         /* pointer to shared tree */
```

`comm` automatically converts all members of a structure to communication variables. It is also possible to declare members of structures explicitly as communication variables to allow other processes to work on several parts of one object simultaneously. This has been done in Program 1 in the declaration of the structure `struct node` for the components `left` and `right` (line 10). Now it is possible for several processes executing on different machines to work concurrently on different branches of the distributed tree with shared pointers to other nodes of the tree.

Every function can be prefixed syntactically by an *interface* (i.e. `(`*interface-list*`).`*function*, see Section 2.3). In classical object-oriented languages, *interface* is used to denote the visible member list of a class, whereas in &Co-languages, the interface enumerates the sharable objects of a programming element (e.g., of a function). The interface consists of a list of variable declarations, separated by semicolons. If the interface of a function is empty, the parenthesis "()" and the dot "." can be omitted entirely. The return values of a function are also passed via the interface. In contrast, an ordinary C function can return only one value which can be a structure (disregarding the use of reference parameters, that can be used to "share" data). Objects that are passed in the interface are used for communication during the lifetime of concurrently running processes.

Every variable can be created dynamically with the function `new`. For example,

```
(int foo).new
(struct node *ptr).new
(comm struct node root).new
```

creates objects of the specified types anywhere in a program (not only at the beginning of a block as in C), e.g., `for( (int i).new = 0 ; i < 10 ; i++ )`.

2.2. Concurrency

An explicit process is created with the function

`(` *where* `;` *func* `;` *pid* `;` *type* `).process`

It calls the function *func*, executed by the C&Co system at the site *where*. The process runs as a separate thread and may communicate with its caller via communication variables that are passed through the interface of *func*. The `process` construct does not wait and the program continues with the next statement.

where is the name of the instance and its Internet address (e.g. `cco@some.site`) and—depending on the architecture—the processor where the C&Co process is to execute. If *where* is `LOCAL` the process is started as another thread at the local C&Co system, if *where* is `REMOTE`, the process is sent to any other C&Co system (depending on the work load of the system).

func specifies the C&Co function to be executed and its arguments. It is an expression of the form:

`(` *interface-list* `).`*funcname*`(` *parameter-list* `)`

Communication variables which are specified in *interface-list* are shared between the concurrent processes.

pid is a communication variable that identifies the new process. It is automatically assigned the value `SUCCEEDED` after the completion of the process, if the called function was not a transaction (see Section 2.3). If `process` was used to call a transaction the assigned values are `SUCCEEDED` after a successful execution of the transaction or `ABORTED` otherwise. The function

```
( pid ; signal ).signal
```

sends signals, e.g., `ABORT`, `STOP`, `CONTINUE`, `MIGRATE`, and `REDO`, to the specified process.

type is either `DEP` (dependent) or `INDEP` (independent). The caller of independent processes does not rely on their termination state. An unfinished independent process is automatically restarted with its previous image after a system failure and can be considered as a software contract between the C&Co-language and the local coordination kernel. An issued `INDEP` process will be eventually executed. Dependent processes must be re-started explicitly by recovered `INDEP` processes (see software contracts in Section 5.3).

2.3. Transactions and Commitment

Transactions are defined in C&Co with the keyword `trans` in two possible ways.

1. Every function can be statically defined as a transaction with

   ```
   trans ( interface-list ).function-name( parameter-list )
       {
       statements
       } ( retry-condition ; prepare-phase ; compensate-action ).commit
   ```

   The function must not have a return value, i.e., it must be of type `void`.

2. Statements can be grouped together to form a transaction with

   ```
   trans
       {
       statements
       } ( retry-condition ; prepare-phase ; compensate-action ).commit
   ```

Every function can be used as an entry point for external processes. It may only access (and return) communication variables which are passed in its interface and therefore cannot interfere with other programs running at that C&Co site. Thus, the `main()` function of C programs becomes superfluous—we just keep its functionality to support the reuse of traditional C programs. An essential feature of communication variables in the *interface-list* is that they are passed by reference (comparable with variable parameters in Modula-2) in contrast to the variables in *parameter-list* which are passed by value, as is the usual mechanism in C.

`commit` corresponds to the end of the transaction and implicitly performs the *commit procedure*. All communication variables are written in an atomic step. The commit procedure waits for all dependent processes (with type `DEP`) to complete.

Transactions are aborted by calling the primitive `abort` which corresponds to a break from a block. This abort causes the compensate actions of all committed subtransactions, but not of sub-subtransactions to be executed. If a transaction fails, i.e., it cannot commit, and the *retry-condition* evaluates to `false`, execution continues immediately after the transaction.

All arguments of `commit` are optional—comparable to the syntax of `for(;;)`. The parenthesis can be omitted entirely if the *interface-list* of `commit` is empty.

retry-condition specifies if the transaction will be retried if it was not successful. More precisely, if the transaction cannot commit, the boolean expression *retry-condition* is evaluated. If it evaluates to `TRUE` (non-zero), then the transaction is retried. If *retry-condition* is omitted, `FALSE` is assumed as the default. A retry value of `TRUE` means repetition until the transaction succeeds. To retry a transaction is reasonable because the environment might have changed in the meantime, e.g., another passenger might have returned his ticket for a flight.

prepare-phase is a function or a process, that is called when it is certain that the transaction can commit.

compensate-action specifies a function which is called *after* the transaction has committed but must be compensated. Only the compensate action of the outermost transaction is called. It is responsible to semantically undo the effects of the transaction and of all subtransactions. Thus, compensation is not a cascading operation. Note that if the signal `ABORT` is sent to a process, that has already finished and did execute a transaction, then its compensate action (if existent) is started. In all other cases all sub-processes and subtransactions of the process are aborted.

All statements between "{" and "}" belong to the scope of the transaction. Global variables and pointer references to local memory are meaningless for functions called via `process`. Only the declarations of types and functions, the values of the variables in *parameter-list*, the communication variables in *interface-list*, and the locally defined ordinary and communication variables are in scope.

The values of communication variables that are not yet committed and are visible within the scope of the transaction can be over-written there, because the variables are treated as local objects. Thus, the rigorous single assignment property only applies at the end of the transaction. The value which has been assigned to the communication variable in the scope of a transaction is exported upon commitment and can never be changed again.

The function

(*comm-obj*)`.undefined`

which may only be used in a transaction, tests if the communication variable *comm-obj* is undefined and checks that the object is still undefined at the point of commitment of the transaction.

The function

(*comm-obj*)`.defined`

tests if the communication variable already has a defined value without waiting for the actual value. Any other attempt to read the value of a communication variable waits until the communication variable is defined by a concurrent thread (see Section 2.4 below).

2.4. Synchronization

Synchronisation is done implicitly using communication variables in expressions. The semantics of "`x = expr`" must be extended for communication variables. The operator "`=`" can be interpreted as "read the value of `expr` and then write this value into `x`". All objects on the right side must be defined before execution can continue. If an undefined communication variable is evaluated (using logic, arithmetic, relational, or bit operators), the system waits until a value is written into the communication variable (by another transaction).

The assignments of all communication variables that have been made within a transaction become globally visible at the end of the transaction (see Section 2.3), provided the communication variables have not received any value yet by another committed transaction. If `x` has already been committed, even if `expr` equals the value of `x`, the assignment fails, due to the rigorous single assignment property of communication variables. In this case, the programmer can only test for equivalency using "`==`".

2.5. Implementation

The coordination kernel was developed at the TU Vienna and runs on the IP network layer. The communication objects are implemented with replication techniques. All replicas are maintained consistently by means of a primary copy migration protocol [31] which is resistant against system and network failures. We are currently adding garbage collection which is crucial, because of the write-once property of communication objects. On the other hand, it is exactly this property which allows an efficient implementation and the design of deadlock free protocols. Language approaches based on shared, updateable objects suffer from the same concurrency control problems as found in classical database literature.

The implementation of a coordination language relies on the coordination kernel. Our pre-compiler of C&Co [23] transforms all accesses to communication objects and all process calls into the corresponding kernel functions. This pre-compiler can treat all communication, concurrency, and transaction properties as described above. Its only limitation is that it does not support intra-process concurrency, i.e., it supports the spawning of processes between several C&Co systems, but not of threads within the same C&Co system. In other words, a `LOCAL` process causes the activation of another occurrence of the C&Co runtime system. As a second limitation, the pre-compiler cannot support all of the dynamic features for object-orientation. (We do not explain the object-oriented features of C&Co because of the limited space but refer to [11], [32] for details.) These limitations can only be overcome by a new C implementation that reorganizes C's internal stacks and provides light-weight processes. This was done in our implementation of $^{\mathrm{V}}\mathrm{P}_{\mathrm{L}}$; it supports fine-grained concurrency.

3. Programming Languages with Work Flow Capabilities

In this section we will summarize general purpose programming languages that offer coordination properties. Languages which have been designed especially for work flow management or for the representation of advanced transaction models are discussed in other articles in this issue.

The notion of a *coordination language* was coined first in [15], where a language is considered to consist of two disjoint parts: a computation part and a coordination part. The latter one can be added to the former one in an orthogonal way. The Linda model [14] is a coordination language in this sense. At almost the same time the term *mega programming* was coined by [50] with similar meaning. The task of a mega programming language is to coordinate preexisting, large software (mega) modules. It must initiate a computation at another site, pass parameters to the remote task, control it, and receive results in a reliable way. Furthermore, a mega programming language should provide partial failure recovery. Due to their autonomy, local systems may fail individually. The mega programming language should allow the substitution of failing services (this relates to function replication in the Flex Transaction model). As coordination of mega modules involves programming in the large, object-orientation is a must [49]. Not only is the modularization of data and names a necessity, but also the integration and modeling of autonomous data and schemes requires concepts like inheritance and polymorphism. Similarly, many object-oriented approaches can be found to integrate data semantically in multidatabase systems [10].

Crucial features of a coordination language which can also be used for work flow management are: linguistic support to communicate and synchronize concurrent activities, explicit specification of inter-process concurrency, and object-orientation.

Many languages and systems exist that aim at one or more of these coordination properties. In particular, a large number of languages and systems that support distributed computing (i.e., communication, synchronization, and concurrency) have emerged over the last ten years [4]. [1], [17], [41], [47] describe parallel and concurrent systems that also aim at object-orientation.

In the following we will briefly classify languages that offer at least support for distributed computing. Such a list never can be complete, as there exist hundreds of such new languages [4]. For many of these languages the authors did not explicitly claim that their languages are coordination, mega programming, or work flow languages.

The design dimensions go from the invention of a completely new language to the creation of library functions that can be called from any language or software system. To extend an existing language by coordination properties stands somewhere in between. We may further differentiate languages by their paradigm, for example the imperative paradigm like C and Pascal, or the 5^{th} generation paradigm like logic, functional and object-oriented programming.

3.1. Classification

Any of the desired work flow features can be programmed with a general purpose tool, but the follwing classification referes only to the pure (original) models of the presented tools. The required features must be part of the model ("built-in"). Table 1 gives a classification of software support tools (in alphabetical order) by:

Shared The tool supports communication based on the shared data paradigm. The idea behind the shared data model is to make replication of shared data transparent to the programmer [3]. She need not care for object location but uses the object as if it is a local one. The implementation takes care of a consistent and exclusive access of the replicated data.

Library The tool provides library functions.

CM The tool provides advanced control mechanisms (see Section 1.1.1). Concurrency control is a necessary prerequisite of a work flow tool and we implicitly assume that every presented tool offers explicit constructs to express and synchronize parallel tasks. However, CM means the tool must provide a declarative formulation and checking of constraints, dependencies, and alternatives (indeterministic specifications). This is usually only possible in higher-level programming paradigms (like for example logic). It can for example be specified by means of guards or other advanced synchronization mechanisms (like implicit blocking on read). Low-level constructs like threads or explicit barriere synchronization are not meant with CM.

SI The tool offers support for system integration (see Section 1.1.2). Naturally all C based/embedded tools, and thus all libraries, offer interfaces and can be called from arbitrary languages. SI includes also systems that support objects that can be shared over system boundaries (offering an explicit object description language, or automatic conversion to the correct data types), or that offer "black box" objects where any kind of data can be transported and it is up to the receiver to understand the communicated data. Many high level languages do not provide SI: they can only communicate with (remote) software systems of their kind.

OM The tool is based on an object model (see Section 1.1.3).

TC The tool has built-in transaction capabilities and thus provides recoverability (see Section 1.1.4). The support of atomic actions is not sufficient: the tool must offer fault-tolerance.

3.2. Libraries for Coordination

CORBA [19] supports the administration of distributed objects. Object data and methods must be described in the IDL (Interface Definition Language) that aims towards object-orientation, similar to the C++ language.

PVM [26] is a library that provides functions to receive and send messages and to spawn remote processes by taking care of the correct data conversions between

Table 1. General Purpose Software Support for Coordination

	Shared	Library	CM	SI	OM	TC
Argus	×	–	×	–	×	×
Arjuna	×	×	–	×	×	×
Concurrent Eiffel	×	–	×	–	×	–
Concurrent Logic	×	–	×	–	–	–
Coordination Kernel	×	×	×	×	×	× (Flex)
CORBA	–	×	–	×	×	–
ISIS	–	×	–	×	–	–
Linda	×	×	×	×	–	–
MeldC	×	×	–	×	×	–
Orca	×	–	×	–	×	–
PVM	–	×	–	×	–	–
RPC	–	×	–	×	–	–
&CO Languages	×	–	×	×	×	× (Flex)

different architectures. PVM's functions are more convenient to program with than sockets, for example.

ISIS [8] is a library that supports reliable broadcast utilities, and group based communication mechanisms based on RPC.

In summary, CORBA, PVM, ISIS, and operating system facilities [46], [48] (like remote procedure call (RPC), sockets, and threads) are too low level to be used as work flow languages by themselves. They can serve as the implementation tools for other work flow languages but do not exhibit satisfying linguistic support for coordination. These libraries introduce a type of thinking in the term of message passing which is non-problem related and does not provide a sufficient abstraction for the programmer. In contrast, the shared data paradigm offers—at the same level of expressiveness as the message passing paradigm—a much better conceptual model. The programmer sees a globally shared object space that serves for communication and synchronization purposes. All other mentioned tools support the shared data model.

MeldC [28], [29] aims at a uniform design so that the reflection about MeldC programs (i.e., the meta-reasoning about programs) can be supported. MeldC is implemented as a runtime object library and supports different kinds of message passing as well as different kinds of threads. It is dynamically expansible by means of dynamic composition, using shadow objects.

Arjuna [20], [47] provides transactions on distributed objects. Arjuna is an object-oriented programming system that itself is organized by means of classes. Replication techniques are used to provide fault-tolerance. Arjuna is a C++ library: its class definitions can be inherited from C++ programs. There are classes for concurrency control and the management of object locks, etc. Arjuna objects are persistent and can be recovered after system failures.

Linda [14], [15] provides blackboard communication and follows the shared data paradigm. More precisely, Linda is a new communication model that should be termed shared data structure paradigm. The basic idea of Linda is a globally shared *tuple space* into which concurrent processes can insert tuples, and from which they may read or delete tuples. Reading and deleting are blocking primitives: if no suitable (matching) tuple can be found, implicit synchronization takes place, analogous to the implicit waiting in C&Co.

This linguistically simple model is expressive and allows the elegant specification of most classes of concurrent problems. The limitations of Linda are: its original design did not consider transactions, i.e., it is not possible to perform several Linda operations in one atomic step, and tuples have global names thus introducing a name space problem. Security might become a problem, because any process that connects to the tuple space may "guess" the name of a tuple which may result in unauthorized access. Moreover, garbage collection is impossible, because there is no point in time when the Linda system can decide that a tuple will no longer be referenced. Fault-tolerance has not been considered, thus a system or network failure might compromise the tuple space.

Many attempts have been made to overcome these limitations [2], [27]. However, as more functionality is added to a sound model, its design becomes less pure.

The Coordination Kernel [12], [31] offers the shared data paradigm by means of communication objects. It provides advanced transactions and thus reliability and software fault-tolerance. Inter-process concurrency is provided by the `process` primitive which can be seen as a reliable software contract. Efficient implementation is assisted by the read-once property of communication objects. Garbage collection is possible, because the kernel can determine, if a communication object is no longer referenced. The coordination kernel offers a set of library functions that can be called from any software system. Among them are the starting, aborting, and committing of transactions, the creation, reading, writing, and testing of communication objects, and the spawning of processes.

These functions can either be called directly from a language, or be integrated more smoothly into the host language, like in C and Prolog, without violating the original language's programming style and by adding only a few language constructs. Motivated by multidatabase systems, it follows the principles of respecting the local system's autonomy which includes not to change or destroy a local system.

A communication variable in a programming language which does not provide object-orientation becomes a typed communication object of the coordination kernel; otherwise the declaration of an object as a communication variable makes the object a communication object which can then be used interchangeably in the host language and by the coordination kernel.

3.3. New Languages and Language Extensions

Argus [38] is a fault-tolerant, object-based programming system that also can be considered an operating system. For recovery the programmer may provide a so-called recovery section that is automatically called by the system after a failure is encountered.

Concurrent Eiffel [41] extends the object-oriented programming language Eiffel by concurrency by introducing a few language constructs. As usual in concurrent and distributed object-based systems, objects communicate by invoking each other's methods.

Concurrent logic programming languages [45] change Prolog' semantics to support explicit parallelism. Representatives of concurrent logic languages are Concurrent Prolog, Guarded Horn Clauses, and Parlog, which use shared logic variables for communication. However, these languages do not transgress the borders of a system and cannot deal with failures, which makes them unsuitable for heterogeneous environments. The approaches to exploit parallelism in Prolog implicitly are not of interest for coordination languages that need explicit language constructs to reason about the data and control flow of activities.

Orca [3], [5] is a new object-based programming language that uses the shared data communication paradigm. Orca provides atomic actions on distributed objects. There exist very detailed performance analyses of Orca implementations in the Internet, on a multi-processor architecture and on the distributed Amoeba operating system, demonstrating that the shared data paradigm is not necessarily less efficient than message passing.

&Co-languages [24], [30], [34] are based on the coordination kernel. C&Co and $^{V}P_{L}$ support multiple inheritance, encapsulation of data and behavior, and different forms of polymorphism ([11], [32]).

The coordination kernel integrates particularly well with Prolog, resulting in $^{V}P_{L}$, because Prolog variables already offer a single assignment property. Like in C&Co, accessing communication variables is through the use of (unification) operators. Concurrency fits well to the logic programming model and can be realized by sequential/parallel AND/OR operators. Transactions are determined by the commit operator "|". The nesting of transactions corresponds to the possibility of nesting goal calls. Testing execution states is completely implicit: this is reflected by the success or failure of goals and causes either forward or backward execution. The process primitive for inter-process concurrency is supported in an analogous way as in C&Co.

3.4. Summary

A completely new language has the advantage of making possible a good language design. In contrast, the addition of a library usually does not integrate with the host language in a satisfying way. The approach to extend a language offers the potential to integrate the extensions in a smooth way, but in many cases the extensions are simply put on top of the language (e.g., Linda adds the new sphere of the tuple space), or destroys or violates the semantics of the host language (e.g., Concurrent Prolog disables Prolog's backtracking mechanism).

Thus, completely new designed languages on the one hand are advantageous, but the problem is that programmers are forced to switch to a new language even if such a language fulfills all requirements in an optimal way. The reuse of existing program components might become difficult.

4. Planning a Trip: A Work Flow Example

The example handles the planning of business trips for university members and is based on the work flow example in [6]. First, we explain the problem and then show its implementation in C&Co. An informal description is given for each example program, assuming that there are no failures and that the program makes good progress, and then possible error cases are analyzed.

We assume, that every member of the university and every office is a participant of a network and has an account on some machine connected to other machines. Moreover, it is reasonable to imagine that he or she has some kind of calendar and a "to-do-list". The to-do-list of all members is managed by the university's computer system and its root is shared at all sites so that anyone can insert tasks into another university member's to-do-list. The to-do-list might be a simple data structure like

```
typedef struct to_do_entry {
    void *task;
    int result;                              /* result value or undefined */
    comm struct to_do_entry *nextitem;
} ToDoEntry;

typedef comm struct {
    comm ToDoEntry *professor;
    comm ToDoEntry *secretary;
    comm ToDoEntry *chair;
    ⋮
} ToDoRoot;

ToDoRoot to_do_list_root;
```

or complex objects described within a class with many methods which display the items of the to-do-list in many different forms. The employee is then able to work with this item, i.e., complete it or postpone it. The result of his decision is written to `result` which might trigger another process or action which is waiting for the completion of that task. Analogous, the insertion of an entry into a foreign to-do-list may cause a new window to appear on the screen of the other university member that displays the task to be done. We will use the simple to-do-list described above in the following sections.

4.1. The Work Flow

A member of the university wants to take part in a conference. He wants to book a flight but needs permission of the department office before he can do so. He makes his request at the office and, provided that the answer is "yes", makes a reservation

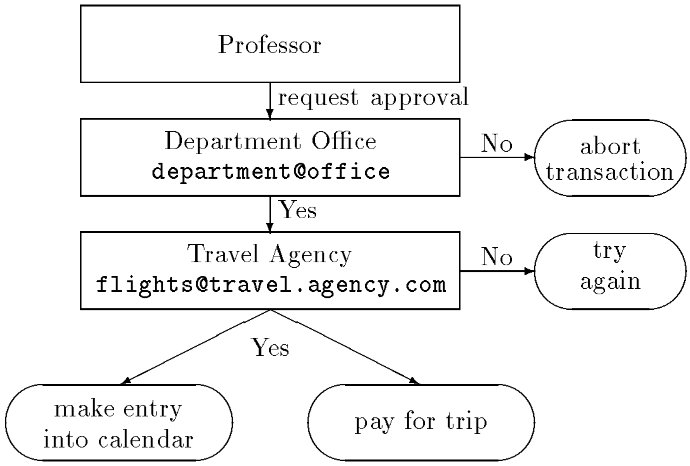

Figure 1. Work Flow at the Professor's Office

at the travel agency. Again, if he gets his flight, he makes an entry into his personal electronic calendar and pays for the trip, or else the transaction is aborted. The work flow is shown in Figure 1.

The professor fills a form with the data of his trip. The data is stored in the communication variable `trip`. The structure of `trip` is shown in Program 2. The professor writes the components `name`, `position`, `purpose`, `dates`, and `cost_estimate`. The other members of the structure are written by other transactions (see Section 4.3). Then he starts a transaction to plan the trip. The structure `Travel` (lines 5–10) is also declared with `comm` because it is passed to the travel agency in line 45 of Program 3 as a separate communication variable.

Program 2 (*Structure of* `trip`)

```
1   typepdef struct Trip {
2       char name[80];
3       char position[20];
4       char purpose[30];
5       comm struct Travel {
6           Dates dates;
7           SCHEDULE schedule;
8           int actual_cost;
9           int reservation_ID;
10      } travel;
11      struct Account {
12          int fund;                      /* LOCAL, GRANT or NONE */
13          char number[20]
14      } account;
15      int cost_estimate;
16  } TRIP;
```

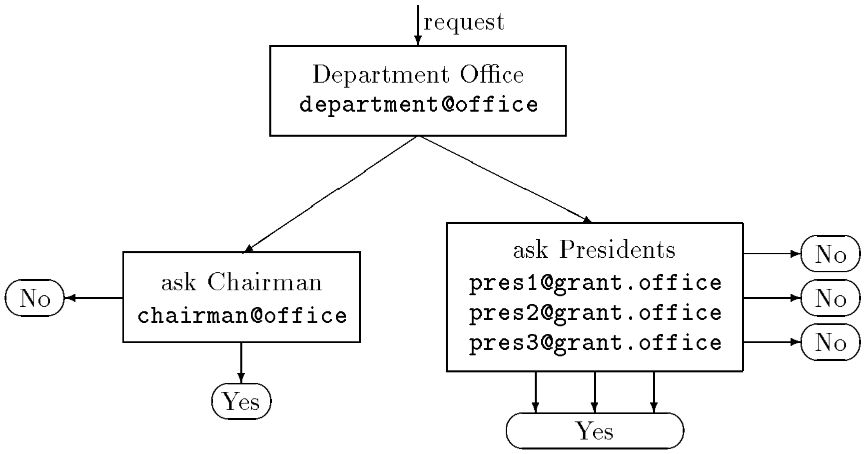

Figure 2. Work Flow at the Department Office

The work flow at the department office is shown in Figure 2. The department office makes two requests for money in parallel. The chairman of the department is asked for money, that comes from the university, and the grant office is asked to provide external funds for the flight. If the money comes from local funds it is sufficient for the chairman to permit the trip. On the other hand, if external money is used, the professor must get the authorization of all three presidents of the grant office. If one of the presidents is against the trip, the professor cannot make a reservation. We do not assume that the grant office shares to-do-lists with the university.

4.2. The Professor's Office

The work flow program at the professor's office is shown in Program 3.

4.2.1. Successful Control Flow

The work flow at the professor's office creates the communication variables `trip`, `PIDapp`, and `PIDpay` (lines 20–22). The professor writes the required data of the travel into `trip` and commits these values in a transaction in lines 23–29. Then he starts a process to carry out the planning of the trip (line 30). The process is started as an independent process (`INDEP`) to assure that it will be performed eventually. The communication variables `trip` and `to_do_list_root`, that represents the anchor to all to-do-lists of the university, are passed to the process via the interface of the function `plan_a_trip`. The deadline for the reservation is given as a value parameter. The communication variables `PIDapp` and `PIDpay`, that are used in lines 43 and 50, must be passed to the transaction to support a correct failure recovery. This is described in more detail in Section 4.2.2.

At first, `plan_a_trip` tries to get money for the trip at the department office. It calls the function `approval` as an `INDEP` process (line 42–43). The test of `trip.account.fund` in line 44 waits until one of the transactions of Program 4 has written a value into the communication variable. If the department office was able to organize the money, the travel agency makes a reservation (lines 45–46). Otherwise the transaction is aborted. The component `travel` of the communication variable `trip` is passed to the process at the travel agency. If the function `reservation`—we don't show its implementation because it is an existing, external program, that we have to reuse—is successful, it writes the still undefined components of `travel`. Because the reservation transaction is started as a dependent process (`DEP`) of `plan_a_trip` the commit in lines 48–52 of the parent transaction depends on the successful completion of `reservation`.

The transaction specifies a retry condition (lines 48–49) and a prepare phase (lines 50–51), but no compensate action. Let us assume that everything works as planned and funds as well as the travel arrangements could be accomplished: the prepare phase is executed, which issues the payment for the trip at the travel agency and inserts the trip into the calendar of the professor.

Program 3 (*Professor's Office*)

```
20  comm TRIP trip;
21  comm int PIDapp;                                  /* PID for approval */
22  comm int PIDpay;                              /* PID for pay_for_trip */
23  trans {                                  /* enter name, position, ... */
24      strcpy(trip.name, name);
25      strcpy(trip.position, position);
        ⋮
29      } commit;
30  (LOCAL; (trip; PIDapp; PIDpay; to_do_list_root).plan_a_trip(some_date);
31      (comm int pid).new; INDEP).process;
    ⋮
40  trans (comm TRIP trip; comm int PIDapp, PIDpay;
41      ToDoRoot root).plan_a_trip(time deadline) {
42      (department@office ; (trip; root;
43          (comm int p[7]).new).approval(); PIDapp ; INDEP).process;
44      if(trip.account.fund != NONE)             /* Synchronisation point */
45          (flights@travel.agency.com; (trip.travel).reservation(trip.name);
46              (comm int PIDflight).new ; DEP).process;
47      else abort;
48  } ( (trip.account.fund != NONE) && (PIDflight == ABORTED)
49      && (actual_time() < deadline) ;
50      (department@office ; (trip).pay_for_trip() ; PIDpay ; INDEP).process,
51          (trip).insert_in_calendar() ;
52      /* no compensate action */ ).commit
```

4.2.2. Control Flow Under the Assumption of Failures

If the transaction cannot commit this can have two reasons: (1) no funding could be organized, or (2) the flight could not be booked. In case (1) the decision is irreversible so that the professor must resign the trip. In case (2), if the fund was given, there is the possibility that the flight can be booked later. Usually, flight reservations are compensated easily because it is always possible to return an already booked ticket. There is the possibility that some passenger returns his ticket, so a flight could become available later. Thus, the retry condition states that if there is money available and if the date until the flight must be booked (denoted by the ordinary variable `deadline`) is not exceeded, the transaction is tried again.

The program is restarted with the same image after a crash. For that reason it is possible to restart the same independent processes `approval` and `pay_for_trip` because they are started with the same PIDs (i.e., `PIDapp` and `PIDpay`)

In case of a retry, the process at the department office is not executed again: by calling it with the same PID (`PIDapp`), the kernel recognizes that this process is still running or has already completed and does not restart it, but reports this fact to C&Co. C&Co in turn skips the process instruction and proceeds with line 44. Depending on the value of the communication variable `trip.account.fund`—which might be already defined—the reservation at the travel agency will either be retried or not called. In case of a retry another process is started at the travel agency that has nothing in common with the previous one, because dynamically a new PID is generated for it.

4.3. The Department Office

The program at the department office—running on a separate site—is shown in Program 4. The department office is responsible for the validation of the travel request.

4.3.1. Successful Control Flow

The function `approval` first generates a new to-do-list entry (called `request`) and writes the data of the trip into its task (lines 103–106). This request is inserted into the to-do-list of the chairman who eventually will process the new request (lines 107–108). The implementation of the function `insert_to_do_entry` is obvious and thus not shown. The process at the chairman office immediately completes. The chairman will sometimes detect the new entry, process it, and place the result in the component `result` of the communication variable `trip`. As a consequence that `trip` is shared between the professor and the chairman, the latter one can supervise at any time what the progress of the professor's travel request is. The chairman may want to know, whether the flight reservation was done, or, if the fund was not a local one, check, whether an external fund was granted. He can do this by simply inspecting the shared communication variable `trip`.

The approval of a fund, either local or external, is a long duration activity and the applicant wants to have the fund as soon as possible. Thus, to improve the response time, the department office applies in parallel for an external grant. The communication with the external funding organization cannot be handled via a shared to-do-list, but is done via the function `travel_grant` (this function is not shown). The communication variable `trip` is passed to `travel_grant` via its interface and thus becomes shared at the grant office. More precisely, a process is sent to each of the three presidents (lines 109–111).

Afterwards, three processes are started for synchronizing the above processes (lines 112-114). The first one checks for a local fund, the second one watches for an external fund, and the third one reports if there is no fund at all. Everyone of the three processes competes to write `LOCAL`, `GRANT`, respectively `NONE` into the component `account.fund` of the shared communication variable `trip`. The result of the fund application is thus represented by `trip.account.fund`.

The function `wait_for_local` is defined in lines 116–122, `wait_for_grant` in lines 123–130, and `wait_for_none` in lines 131–136. As they try to write the same communication variable, only one of them can succeed, all others must fail, because a communication variable can be written only once.

Let us assume that `wait_for_local` succeeds, i.e., the local fund has been granted which is reflected in the to-do-list entry of the chairman (`request.result` is tested to have the value `YES`, line (118)). The transaction `wait_for_local` locally writes `LOCAL` into `trip.account.fund` and reaches its commit. Provided the process executing `wait_for_grant` has not succeeded in the meantime, the commit can be performed and makes the written value visible. Moreover, the commit is accompanied by a prepare phase (line 121) that sends `ABORT` signals to the processes running at the grant office. The sending of the signal `ABORT` to a president's process causes the compensation of all transactions done by this process and hopefully this way causes the compensation of the grant approval. The compensate action which is specified in line 122 sends the signal `ABORT` to the process at the chairman's office. This compensate action is activated, if for example no flight can be reserved in time and the trip planning aborts.

Let us now assume that `wait_for_local` is still running and that `wait_for_grant` succeeds. The success of external funding is checked by testing the PIDs of the three processes at the grant office (we assume that `p[`i`]` reflects the answer of president i ($i = 1, 2, 3$). Now `wait_for_grant` commits and in its prepare phase (line 128) sends an `ABORT` signal to the process at the chairman office. Analogous to above, a compensate action is defined that sends the signal `ABORT` all processes at the grant office (line 129). The function `wait_for_none` is crucial for the case that no fund can be organized. It prevents line 44 in Program 3 to block forever.

Program 4 (*Department Office*)

```
void (comm TRIP trip; comm ToDoRoot root; comm int p[7]).approval( void )
{
    /* ask for local fund and external grant in parallel */
    trans {
        (comm ToDoEntry request).new;
        request.task = &trip;
    } commit;

    (chairman@office ; (request; root.chair).insert_to_do_entry() ;
     p[0] ; INDEP).process;

    (pres1@grant.office; (trip).travel_grant(); p[1]; INDEP).process;
    (pres2@grant.office; (trip).travel_grant(); p[2]; INDEP).process;
    (pres3@grant.office; (trip).travel_grant(); p[3]; INDEP).process;

    (LOCAL; (p; request).wait_for_local; p[4]; INDEP).process;
    (LOCAL; (p; trip).wait_for_grant; p[5]); INDEP).process;
    (LOCAL; (p; trip).wait_for_none; p[6]); INDEP).process;
}

trans (comm int p[7]; comm ToDoEntry request).wait_for_local()
{
    if(request.result == YES)
        request.task->account.fund = LOCAL;
    else abort;
} ( ; ((p[1];ABORT).signal, (p[2];ABORT).signal, (p[3];ABORT).signal) ;
    (p[0]; ABORT).signal ).commit

trans (comm int p[7]; comm TRIP trip).wait_for_grant()
{
    if(p[1] == p[2] == p[3] == SUCCEEDED))
        trip.account.fund = GRANT;
    else abort;
} ( ; (p[0]; ABORT).signal) ;
    (p[1]; ABORT).signal, (p[2]; ABORT).signal, (p[3]; ABORT).signal
    ).commit

trans (comm int p[7]; comm TRIP trip).wait_for_none()
{
    if( (p[4] == ABORTED) &&
        ((p[1] == ABORTED) || (p[2] == ABORTED) || (p[3] == ABORTED)) )
        trip.account.fund = NONE;
} commit;
```

4.3.2. Control Flow Under the Assumption of Failures

Let us now discuss the behavior of this work flow, if failures occur. The trip planning is started as an `INDEP` process (line 30 in Program 3), that is automatically recovered after a failure by the kernel, provided it has not yet terminated. The function `plan_a_trip` tries to revoke `approval`, however, if `approval` has already been started and not yet terminated, it is also automatically recovered by the kernel. Thus, its re-start by `plan_a_trip` has no effect. If `approval` had not yet been started before, i.e., the failure occurred immediately after line 41, the process is now started for the first time. It is thus guaranteed that a professor applies only once for a trip at his chairman.

The process that executes `pay_for_trip` also is not executed twice (for the same reasons as explained above). Once issued, it is sure that the payment will be done, because the process is started as an `INDEP` one. Even if a failure occurs immediately after the execution of the prepare phase and before the commit completes, the exactly once semantics can be guaranteed.

The above described failure behavior can only be achieved, because the PIDs of the processes `approval` and `pay_for_trip` appear in the arguments of the process, executing `plan_a_trip`. Thus, the references to these PIDs can be recovered after a failure. Note that `plan_a_trip` is automatically recovered by the kernel with its old image that includes `PIDapp` and `PIDpay`.

On recovery, the process at the department office will try to re-start the process at the chairman office and the processes at the grant office. If such a process has already been started, i.e., it was invoked before the failure occurred, it is automatically recovered by the kernel running at the department office. If it was not yet started, it is now started on recovery by the re-execution of `approval`. It is thus crucial to recover the PIDs of all these processes. This is the reason why they are passed in the interface of `approval` via the array `p[7]`.

The processes in lines 112–114 need not have an only once semantics. They are started as `INDEP` processes for another reason: not all of them can succeed. If they were started as `DEP` processes, the enclosing transaction could never commit, because a transaction depends on the success of all `DEP` processes.

5. Evaluation of C&Co as a Work Flow Language

C&Co is a general purpose programming language based on C. It is an "engine" that can serve a work flow management tool directly, or that can be used to realize other work flow models and graphical visualization tools for work flow. The user is not restricted to predefined features. As an analogy, the advantage of using VPL as an MDBS language like MSQL [39] is that—besides involving transaction control aspects—any access pattern to the distributed databases can be modeled with VPL, whereas MSQL and other SQL extensions are not computationally complete. Transitive closure and non-first normal form data can easily be expressed in Prolog based MDBS languages [33]. Obviously, CAD or multi-media applications must handle complex, structured data. The following features are provided in C&Co:

Communication is done via communication variables. Any data represented by C variables can be declared as persistent, sharable objects.

Transactions. Any C function can be prefixed by a construct declaring it as a transaction entry. Within a function, statements can be grouped to form a transaction, too. Nesting of transactions is supported.

Compensation, Prepare Phase, Retry are part of the transaction specification. They are optional: if unspecified, no compensation and no prepare phase are executed, and the transaction is not retried.

Concurrency Control is supported by means of the `process` primitive. It spawns a thread at the local C&Co system *(intra-process concurrency)*, if this is supported by the C&Co implementation at hand (else a new invocation of C&Co is triggered), or at another (remote) C&Co system *(inter-process concurrency)*.

Abstract Object Types, Inheritance, Polymorphism are discussed in general for all &Co-languages in [32] and more specifically for VPL in [11].

The following features are of particular interest for work flow specification. They are implicit in C&Co and can be derived by means of the above explicit language constructs following the principle that a desired feature need not necessarily result in a new language construct:

Execution State Testing is done by investigating the `pid` of a process. It can either be `SUCCEEDED` or `FAILED` and is set by C&Co appropriately at the end of the process. The `pid` itself is a communication variable: it can either be read or tested. If a process did not yet terminate, a testing thread is blocked until the `pid` is set. The programmer does not have to check the `pid` of a dependent process explicitly. The execution states of all dependent processes are automatically examined as a part of the commit procedure of the enclosing transaction.

Synchronization, Data Exchange. The rationale behind data exchange and synchronization in &Co-languages is that local memory is conceptually expanded to the memory of distributed computers in a network. The programmer treats local and shared variables equally. However, the access time to a shared variable may be longer. Reading of an undefined shared variable blocks (implicit synchronization).

The interface of a function allows more general data exchange patterns than in traditional imperative languages. Usually, parameters are passed between caller and callee at two points in time: first, the actual parameters are passed when a function is invoked, and second, the return data are passed back at the function's end. Data exchange in &Co-languages can take place via any communication object that appears in the interface during the whole lifetime of the processes that share data. In contrast to the client-server based message passing mechanism the communication object mechanism provides a symmetric way of communication. The processes agree on the objects they want to share and access the variables in a symmetric way. Many problems do not decompose into client-server naturally but can be solved more conveniently with shared data communication.

Branching, Repetition. Branching obviously follows the same control facilities as C. Repetition of a transaction specification can be achieved by the retry function that is part of the transaction definition, or by the other control constructs of C.

Dependencies. *Data dependencies* state the dependency on the results (effects) produced by other concurrent activities. This can be modeled by means of communication variables. *Execution state dependencies* have been discussed above. *External dependencies* like a constraint that a process must be started at a certain time, or constraints that are derived from the environment (e.g., availability of operating system resources) can be programmed in the C subset of C&Co.

Contracts are signed by means of recoverable processes.

Nesting, Visibility, Granularity, Delegation [18]. Nesting is achieved by the static or dynamic composition of transactions. The transaction's commit makes writings to communication variables visible in one atomic step and gives gives the programmer control over the visibility of communication variables (i.e., commit granularity), as well as the granularity of the atomicity.

Tracing, History. Thanks to their write-once property, all values assigned to communication variables remain forever. This can be seen as a history of all committed effects. Tracing and version control are therefore straightforward tasks.

5.1. Uniform Language Design

The language design of C&Co was driven by the principles of: reflectivity, hardware abstraction, reuse and generalization. C's semantics are not destroyed by C&Co but only slightly expanded: seven new language constructs have been added for communication, concurrency and transactions. Seven more are required for object-orientation. Original C programs can incrementally be upgraded towards C&Co. Existing code that need not undergo parallelization or distribution can be reused without any change.

Reflectivity. The C&Co language consists of three types of programming elements [30]: classes, functions, and types. They are treated in a uniform way. Every programming element can be pre-fixed by an interface. The interface of a class is used for inheritance. The interface of a type serves parameterization; it enables programming with generic programming elements (*universal, parametric polymorphism* [13]). The interface of a function forms its "window" to the globally shared space (see Section 2.1). Functions with the same name can be differentiated by their interfaces (*ad-hoc polymorphism through overloading* [13]). Like in MeldC [29] a reflective architecture is provided: the entire C&Co system can be described by means of meta-classes. This is particularly easy in VPL, as VPL, as a logic based language, provides meta-reasoning.

Hardware Abstraction. Inter- and intra-process communication need not be differentiated: Communication objects form one uniform mechanism for the communication between threads on the same site and between distributed processes. Many other languages clearly differentiate between internal and external communication so that the underlying hardware is reflected in the program. In contrast, C&Co provides a high level of abstraction from the architecture.

Reuse. All programming elements of C can be reused in C&Co without change. More functionality is added by prefixing them with an interface, the `comm` type qualifier, or the keyword `trans`.

Generalization. Instead of introducing a new linguistic construct for every desired feature, the &Co-languages aim at generalizing existing principles. As explained above, data sharing is understood as an abstraction of the own memory to the memories of distributed computers. Parameter passing is generalized by means of the interface. Functions are entry points for (remote) process requests (this is a generalization of the `main()` function, making `main()` superfluous). Communication objects may be used for any kind of communication, thus they also generalize input/output in a sound and reliable way: `stdin` and `stdout` can be substituted by communication objects that represent streams. The notion of an `AOT` *(abstract object type)* [32] generalizes the `void` abstraction for C data types to classes and functions which enables programming with generic programming elements (comparable with *templates* in C++).

5.2. Interoperability

All languages that belong to the family of &Co-languages are interoperable [24]. Even if the languages belong to different paradigms, they can share objects with and spawn processes on software systems that "speak" another language. The coordination kernel is in charge of correct language and data format translations (also between different hardware platforms). We have developed mappings between Prolog and C data types: for example, a Prolog data structure can be set in equivalence to a C data structure, if it is defined how the notions of *arity* and *functor* are to be represented in C. Thus, coordination contributes to unify language paradigms, suppressing the need to build complex, multi-paradigm languages.

As a consequence, C&Co can serve for both task and work flow specification. In work flow applications, the integration of existing software modules is a must. For example, there exist several mailing facilities written in C. Almost all software systems, e.g., Oracle, provide an interface to C. The interfacing of C&Co with Oracle is therefore straightforward. If the local system does not offer a C interface, an interface can be built using pipes or files. A Prolog-Oracle interface has been built by our group, too [44], for the VPL based multidatabase system. VPL can serve for the specification of interactive work flow patterns ("dynamic", "ad-hoc" work flows). The dynamic manipulation of programs is possible because of Prolog's meta-programming capabilities.

The composition of existing components to larger ones, and the specification of the interaction patterns between them, is much easier and efficient in a coordination language compatible to C than in a foreign language. The reuse of existing components is therefore easy.

5.3. Reliability

Reliability comprises the fault-tolerant behavior of object based communication, and the trust that a started process eventually will be done. We show how never-ending processes can be modeled. Unauthorized access to shared data is impossible.

Fault-Tolerance. All visible data of a work flow application must be persistent. Assume that a decision is made, based on data seen in the work flow application.

If these data are lost, the justification for the work done after this decision, is lost, too. In analogy, a global data structure, like a serializability graph employed by a multidatabase system, is crucial. Its loss may cause inconsistencies and compromise the correctness of the global transaction [31]. Communication via communication objects is reliable: committed data are never lost (see Section 2.5). On creation, the communication strategy can be selected for a communication object individually. All the protocols we have designed so far provide reliability, albeit at different degrees of availability.

Software Contracts. Once issued, the coordination kernel is responsible that a `process` will execute. Internally, the coordination kernel writes the process contract into a local communication object, that the user is not aware of (a "process communication object"). This communication object is persistent and thus automatically recovered after a failure. At recovery, the coordination kernel checks all such communication objects and if they represent a process of type `INDEP` the process is automatically restarted. The references to all communication objects appearing in the process's interface are stored on durable medium. After a catastrophic event, the process obtains access to the actual state of its communication objects and can re-start its execution. Depending on the host language in which the process is implemented, a recovered process will re-execute the same statements as before. (In non-deterministic languages there is in general no guarantee that a re-execution will enter the same paths: it may happen that some work cannot be reused on recovery.) Thus it may happen, that a transaction-begin of an already completed transaction is issued a second time. The coordination kernel recognizes this fact by the transaction identifier and informs the caller of the transaction-begin, which in turn will skip this transaction. Analogous, if a process is issued, the `pid` of which is already known to the coordination kernel, the kernel tests whether the `pid` is already defined (i.e., the process has already finished). If not, the coordination kernel will make sure that the process is running (i.e., if it is of type `INDEP`, it should be running already, otherwise it is recovered with its old image).

&Co-languages provide forwards recovery, with `INDEP` process calls as the checkpoints. All committed data are recovered. Uncommitted data as well as read requests, and all requests of still running transactions are lost at failure. Once a transaction has reported its prepared state, its requests are saved and a subsequent `commit` is possible, even if a failure occurs between prepare and commit.

Reactive Activities. Finite computations that map a set of input data into output data without further interaction with the environment are termed *transformational processes*. *Reactive processes* require interaction with the environment and might be perpetual computations [37]. Work flow applications might be long living activities, like complex cooperative design processes, or everlasting transaction, like the supervision of power plants. Even if a failure appears, on a system re-boot, these processes are recovered.

With help of `INDEP` processes, endless running servers can be built. For example, in a &Co-based operating system, there is no "session end": a user simply sends the signal `PAUSE` to the process representing the session and reactivates the process the next time via the signal `CONTINUE`. This process will have in its interface the references to all relevant data it will access. As an example, a local name server could be built with this mechanism that recovers the list anchor to all names it administrates. Thus, the need for global names in a restricted domain can be implemented by the mechanisms provided by &Co-languages. Note, however, that &Co-languages themselves do not provide global names for communication objects.

Security. Communication objects can be accessed only by a process that possesses the right to do so, i.e., the process must either have created the communication object, or it must have been passed the reference to it via the interface. All objects referenced by an object that a process may see, are visible too (e.g., all elements of a linked list). Even if a process "guesses" an object identifier—which should not be possible, anyhow—the access will be refused. Further security is provided by the type mechanism: a type serves as an *access key* and assures that a communication object can only be accessed by adequate methods. This makes sure that a process "understands" communicated data [43].

5.4. Migration of Programming Elements

Processes can migrate by sending them the `MIGRATE` signal. This causes the process to be paused, its transportation to the other site (i.e., the internal process communication object to be shared with this site), and the continuation of the process there. If not otherwise specified, the destination of resources used by the process remains unchanged. The creation of communication objects of type `class` provides the possibility to create sharable class definitions which can be passed to other processes. This enables the reliable transportation of entire program modules. The migration of functions and entire modules (classes) in VPL is particularly easy, because VPL supports a meta-call facility that allows the interpretation of a dynamically composed data structure as a goal.

5.5. Transparency and Metaphors

The shared data communication mechanism provides a conceptually higher abstraction than message passing [5]. The user thinks in terms of a globally shared *object space*. A process possesses a window containing the communication object references it may access. The entirety of all coordination kernels represents the *agent space*. The agents are responsible for the maintenance of the communication objects. Every access must be sent to a local agent—direct communication object access is not possible. Write and read accesses need a final confirmation (commit) of the user before they are done atomically. A process groups a part of the computation to either enable its concurrent execution, to distribute it, or to sign a contract with the agent to ensure the execution of that part. This simple model underlines the conceptual superiority of shared data communication. The abstraction of communication via message sending is much less natural.

As stated above, &Co-languages free the programmer from thinking in terms of the network: reliability is guaranteed as soon as data are committed. Higher availability also does not require knowledge about replication strategies and other implementation techniques. The metaphor for this situation is to have a letter registered or not. This is usually done by adding postage stamps to the envelope, but without forcing the user to care for its transportation. The same holds for communication objects: by selecting another communication strategy when the communication object is created, its availability can be fine-tuned.

Network and system failures are hidden from the user; they are only reflected in longer waiting times, if the chosen protocol does not provide enough availability (i.e., if it does not select another primary copy through voting, but waits until the failing system reboots). Only logical, application related failures are to be masked by the user. For this, function replication can be employed which is easy to specify in &Co-languages.

6. Conclusion

Instead of defining a completely new language or system, we have extended traditional programming languages by coordination properties towards general purpose work flow management languages. The programming style of the host language is not changed. Reuse of existing components becomes easy. The specification of the interaction of the work flow tasks can be done by means of the coordination extensions. These include a high-level, reliable communication mechanism, advanced transactions, concurrency, and object-orientation.

Languages expanded by coordination are interoperable and can cooperate by crossing hardware and language borders.

Acknowledgments

We acknowledge the helpful comments of Manfred Brockhaus, Mehdi Jazayeri, Herbert Pohlai, and Konrad Schwarz on this text.

References

1. G. Agha, P. Wegner, and A. Yonezawa, editors. *Research Directions in Concurrent Object-Oriented Programming.* The MIT Press, 1993.
2. D. E. Bakken. *Supporting Fault-Tolerant Parallel Programming in LINDA.* PhD thesis, University of Arizona, Department of Computer Science, August 1994. TR 94-23.
3. H. Bal. *Programming Distributed Systems.* Prentice Hall, New York, 1990.
4. H. Bal, J. Steiner, and A. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.
5. H. Bal and S. Tanenbaum. Distributed programming with shared data. *Computer Languages Journal*, 16(2):129–146, 1991.

6. D. Barbará, S. Mehrotra, and M. Rusinkiewicz. INCAS: A computation model for dynamic workflows in autonomous distributed environments. Technical Report 97, Matsushita Information Technology Labratory, 2 Research Way, 3rd Floor, Princeton, N.J. 08540 USA, May 1994.
7. Ph. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.
8. K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, December 1993.
9. M. W. Bright, A. R. Hurson, and S. H. Pakzad. A taxonomy and current issues in multidatabase systems. *IEEE Computer*, March 1992.
10. O. Bukhres and A. Elmagarmid, editors. *Object-Oriented Multidatabase Systems.* Prentice-Hall, 1994. in print.
11. O. Bukhres, A. Elmagarmid, and e. Kühn. Advanced languages for multidatabase systems. in [10], 1994. to appear.
12. O. Bukhres and e. Kühn. Highly available and reliable communication protocols for heterogeneous systems. *Information Sciences*, 1995. to appear.
13. L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4), December 1985.
14. N. Carriero and D. Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, 21(3):323–358, September 1989.
15. N. Carriero and D. Gelernter. Coordination languages and their significance. *Communications of the ACM*, 2(35):96–107, 1992.
16. J. Chen, O. Bukhres, and A. K. Elmagarmid. IPL: A multidatabase transaction specification language. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, Pittsburgh, PA, May 1993.
17. R. Chin and S. Chanson. Distributed object-based programming systems. *ACM Computing Surveys*, 23(1):91–124, March 1991.
18. P. Chrysanthis and K. Ramamritham. ACTA: The SAGA continues. Chapter 10 in [22], 1992.
19. Digital Equipment Corporation, Hewlett Packard Company, HyperDesk Corporation, NCR Corporation, Object Design, Inc., SunSOft, Inc. *The Common Object Request Broker: Architecture and Specification*, December 1991. Draft 10, Revision 1.1, OMG Document Number 91.12.1.
20. G. N. Dixon, G. D. Parrington, S. K. Shrivista, and Stuart M. Wheater. The treatment of persistent objects in Arjuna. In *Proceedings of the Third European Conference on Object-Oriented Programming ECOOP89*, pages 169–189, July 1989.
21. A. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A multidatabase transaction model for InterBase. In *Proceedings of the 16th International Conference on Very Large Data Bases*, August 1990.
22. A. K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications.* Morgan Kaufmann Publishers, 1992.
23. A. Forst. Implementation of the coordination language . Master's thesis, University of Technology Vienna, 1994. in preparation.
24. A. Forst, e. Kühn, H. Pohlai, and K. Schwarz. Logic based and imperative coordination languages. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems, PDCS'94*, Las Vegas, Nevada, October 6–8 1994. ISCA, in cooperation with ACM, IEEE.
25. H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the ACM SIGMOD Annual Conference*, San Francisco, May 1987.
26. G. A. Geist and V. S. Sunderam. Network-based concurrent computing on the PVM system. *Concurrency: Practice and Experience*, 4(4):293–311, June 1992.
27. D. Gelernter. Multiple tuple spaces in Linda. In E. Odjik, M. Rem, and J. C. Syre, editors, *PARLE'89*, pages 20–27. Springer Verlag, 1989.
28. G. E. Kaiser and B. Hailpern. An object-based programming model for shared data. *ACM Transactions on Programming Languages and Systems*, 14(2), April 1992.

29. G. E. Kaiser, W. Hseush, J. C. Lee, S. F. Wu, E. Woo, E. Hilsdale, and S. Meyer. MeldC: A reflective object-oriented coordination language. Technical Report CUCS-001-93, Columbia University, January 1993.
30. e. Kühn. Development of software for the coordination of distributed systems. Technical Report TR Project P7773-PHY, University of Technology, Vienna, September 1992. (in German).
31. e. Kühn. Fault-tolerance for communicating multidatabase transactions. In *Proceedings of the 27th Hawaii International Conference on System Sciences (HICSS)*, Wailea, Maui, Hawaii, January 4–7 1994. ACM, IEEE.
32. e. Kühn. A universal model for the coordination of distributed systems. Technical Report TR Project P9020-PHY, University of Technology Vienna, submitted for publication 1994.
33. e. Kühn and T. Ludwig. VIP-MDBS: A logic multidatabase system. In *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems*, Austin, Texas, December 1988. IEEE Computer Society Press. Also included as part of the text of a new IEEE Computer Society Press tutorial *Multidatabase Systems: An Advanced Solution for Global Information Sharing*, by A. R. Hurson, M. W. Bright and S. Pakzad, 1993.
34. e. Kühn, H. Pohlai, and F. Puntigam. Concurrency and backtracking in $\mathrm{V}^{ienna}\mathrm{P}^{arallel}_{\mathrm{L}ogic}$. *Computer Languages Journal*, 19(3), July 1993.
35. e. Kühn, H. Pohlai, and F. Puntigam. Communication and transactions in $\mathrm{V}^{ienna}\mathrm{P}^{arallel}_{\mathrm{L}ogic}$. *Computers and AI Journal*, 13(4), 1994.
36. e. Kühn, F. Puntigam, and A. K. Elmagarmid. An execution model for distributed database transactions and its implementation in VPL. In *Proceedings of the International Conference on Extending Database Technology, EDBT'92*, Vienna, March 1992. Springer Verlag, LNCS.
37. A. J. Kusalik. Specification and initialization of a logic computer system. *New Generation Computing*, 4:189–209, 1986. Ohmsha, Ltd. and Springer.
38. B. Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.
39. W. Litwin, A. Abdellatif, B. Nicolas, Ph. Vigier, and A. Zerounal. MSQL: A multidatabase manipulation language. *Information Sciences*, June 1987. Special Issue on DBS.
40. W. Litwin, Leo Mark, and Nick Roussopoulos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22(3), 1990.
41. B. Meyer. Systematic concurrent object-oriented programming. *Communications of the ACM*, 36(9):56–80, September 1993.
42. M. T. Özsu, U. Dayal, and P. Valduriez, editors. *Distributed Object Management*. Morgan Kaufmann Publishers, 1994.
43. J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4), November 1984.
44. K. Schwarz. SOS—the SICStus/Oracle Selvedge. Master's thesis, University of Technology Vienna, October 1994.
45. E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, September 1989.
46. J. Shirley. *OSF Distributed Computing Environment: Guide to Writing DCE Applications*. O'Reilly & Associates, Inc., 1992.
47. S. K. Shrivastava, G. N. Dixon, and G. D. Parrington. An overview of the Arjuna distributed programming system. *IEEE Software*, pages 66–73, January 1991.
48. A. S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 1992.
49. P. Wegner. Concepts and paradigms of object-oriented programming. In *Proceedings of the OOPSLA-89*, June 1990. Expansion of Keynote Talk.
50. G. Wiederhold, P. Wegner, and S. Ceri. Toward megaprogramming. *Communications of the ACM*, 35(11), November 1992.