Logic Based and Imperative Coordination Languages

Alexander Forst, eva Kühn, Herbert Pohlai and Konrad Schwarz

In: Proceedings of the PDCS'94, Seventh International Conference on Parallel and Distributed Computing Systems, ISCA, IEEE, Las Vegas, Nevada, October 6-8, 1994.

Logic Based and Imperative Coordination Languages*

Alexander Forst, eva Kühn, Herbert Pohlai and Konrad Schwarz
University of Technology Vienna, Institute of Computer Languages
Argentinierstr. 8, 1040 Vienna, Austria, Europe
{alex,eva,herbert,schwarz}@mips.complang.tuwien.ac.at

Abstract

This paper deals with the cooperation between programming systems that belong to different paradigms. We explain our approach towards coordination in the context of the recently proposed notions of mega-programming and coordination languages. We show how our logic based parallel language VPL (Vienna Parallel Logic) can be decomposed into a computation and a coordination part. The language constructs necessary for the coordination of autonomous software systems include a highly abstract communication mechanism, an advanced transaction model and concurrency.

The coordination part is generalized so that it can be embedded into programming languages of different paradigms, for example into imperative languages. This concept resembles the Linda approach in many aspects and we compare the two. A main difference is that instead of adding orthogonal constructs that are in no way related to the host language, we embed coordination into the language so the result is a new language that does not change the programming style of the underlying host language.

1 Introduction

Over the past two years, our group has been working on a parallel logic language called VPL (Vienna Parallel Logic) designed for multidatabase applications (see for example: [5, 15, 16, 17, 19, 20]). The design of this language is nearly complete, first implementations are beginning to emerge.

A new concept in parallel system design is the coordination language, the most important representative being Linda [12]. The creators of Linda view their framework as an orthogonal extension to sequential languages, and indeed, Linda has been grafted on to languages such as C [6], Prolog [4], Fortran and Lisp. The host language is responsible for computational aspects of the program, the coordination language for communication aspects.

A second new concept is mega-programming [22]. It arises from the increasing need to combine existing applications (so-called legacy systems) into larger functional units. This must be done with minimal intrusion into the existing application, either because the application is not accessible to change, or it is infeasible to do so. In multidatabase theory, this is collectively known as the local autonomy of the sites offering these applications [10].

VPL fits well into the framework generated by these concepts. Gelernter and Carriero's [12] proposition that the computational language and coordination language should be completely independent of one another leads us to ask if the communication part of VPL may be used in a different (computational) language context. To this end, we analyzed VPL's communication and coordination features, extracted them and coupled them with the C language, creating what we call C&Co (C and Coordination).

In Section 2 we analyze which language constructs are required to support coordination. In Section 3 we show how these coordination constructs can be embedded into Prolog and C, resulting in the VPL and C&Co languages. In Section 4 we demonstrate by means of an example how languages of different paradigms can cooperate by using the proposed coordination extensions. In Section 5 we compare the proposed ideas to related work.

2 Coordination Concepts

Coordination requires language constructs to control local and autonomous software systems (mega-modules) that are distributed in a network and may run concurrently [22]. A coordination language must be able to pass parameters to local systems, initiate and control computations there, pass results back and combine them. Additionally, if a local system cannot

*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).

fulfill its task, another system may be selected instead (*function replication* [8]).

2.1 Toolkit for Coordination

We propose a *toolkit* for coordination that can be integrated into different language paradigms. The coordination extensions consist of: (1) a reliable and highly abstract mechanism to communicate, (2) language constructs to control concurrency, and (3) a built-in advanced transaction model.

2.1.1 Shared Objects

Message passing and communication via shared memory are communication mechanisms of equal expressiveness. In general, message passing primitives, such as synchronous or asynchronous message passing, RPC, and rendezvous, are low level communication concepts that do not necessarily fit well into the host language and thus do not lead to a clean integrated language concept which meets the requirements of distributed systems.

Shared memory communication provides a higher level of abstraction. All sites participating in the communication have a common view on shared data objects, which have been proposed as the communication medium for many advanced language paradigms (for example, shared logic variables [13, 21], shared data structures [6, 4], shared updateable objects that are written in atomic actions [2], and shared objects providing transactions and fault-tolerance [7]).

We propose therefore to communicate by a mechanism that belongs to the class of shared memory communication, and introduce *communication objects* as communication media. A communication object is just like a normal variable. It can hold the same values and can have the types that are supported by the base language (including structures, lists, sets,...). It can also contain other communication objects as components or point to them, e.g. for infinite lists. The main feature of communication objects is that they are written in transactions and they can be assigned a value only once (*rigorous single assignment property*). Each process can rely on communication objects, because they will not change and are recoverable.

Between processes, communication objects are shared by passing them as arguments (global names are not required). Processes that have not been given a reference to a communication object cannot access it. This mechanism provides security as only authorized processes may access the data. Communication objects provide one uniform mechanism for inter- and intra-process communication. Depending on the underlying architecture, a shared communication object is either held in shared memory or is represented as a distributed tree. Each node of the tree represents one copy of the object. One copy is designated as the *primary copy* which must be owned by a process in order to access it, the other copies are named *secondary copies*. The object tree extends when the communication object is passed to other processes. tree builds up. The structure of this tree reflects the geneology of the object. To read, write, or test an object, a process has to obtain the primary copy. This may change the tree structure because the roles between father and sons might be reversed. The protocol and the application of this reliable communication mechanism in the domain of multi database systems are presented in [16].

2.1.2 Concurrency

Concurrency in a coordination language is needed because the software systems it coordinates already exist and run in parallel on different sites. A language that provides concurrency is the adequate tool to reason about real world parallelism. Although concurrency may be emulated with sequential languages, the level of abstraction provided by concurrent language constructs is much higher.

We differentiate between inter- and intra-process parallelism. The first one is crucial for coordination languages and requires a language primitive (we call it "`process`") to spawn and control a process on another site, and to pass communication objects which become shared between the initiating site and the site where the new process is created.

The computer site where the process is created can be specified. As a default the local site is assumed and the process is executed as another thread of the system by which it was called. This provides a sufficient form of intra-process parallelism for imperative languages.

2.1.3 Advanced Transaction Model

All processes that share communication objects must have a consistent view of these objects. This can be achieved by ensuring that values can be written into the shared space by transactions only. Classical transactions provide the ACID (Atomicity, Consistency, Isolation and Durability) properties and thus the writing of the objects becomes serializable and partial failure recovery can be guaranteed, as communication objects can be recovered after a system crash.

As these communication objects are used for the communication between autonomous systems, we propose to use an advanced transaction model, the Flex Transaction Model [8], that has been designed especially to meet the requirements found in heterogeneous environments. It provides *function replication*, *seman-*

tic compensation and *relaxation of the isolation property* of transactions [3, 9].

We need function replication to be able to substitute a failing service by another one that fulfills the same task in an equivalent way.

Relaxation of the isolation property is important if we assume that we coordinate local database systems. A subactivity at a local database system will require and keep locks there until the global activity completes. The autonomy principle is heavily violated. In particular, long-living, and possibly ever-lasting activities are an unsolvable problem. Therefore, recent work on advanced transaction models for heterogeneous systems has dealt with allowing subtransactions to commit before the complete task is finished.

Semantic compensation is important if we allow the isolation property to be relaxed. In this case a completed action might not be needed (due to function replication) or be aborted (if the global task fails). A successful transaction must not be rolled back, because other autonomous processes may already have seen its effects and may have based their computations on them. Thus, if a process later on decides that a communicated value is no longer valid, it cannot simply over-write this value, but must *semantically compensate* it. This means that a user-defined compensate action may be specified, which is automatically activated. In summary, only actions that are compensatable may complete early.

The necessary language extensions to support the Flex Transaction Model include constructs to express:

1. **transactions.** The start and completion (commitment) of a transaction in the programming language must be made visible. All communication objects are written upon commitment in an atomic step and made globally visible.

2. **write/test/read of communication objects.** Write operations are collected until the transaction on the communication objects is committed, their values become visible in one atomic step. If a communication object cannot be written, e.g., because it has already been assigned a value by another transaction, the transaction fails. Test operations serve to examine the state of communication objects, i.e, whether they still are undefined (free/unbound) or defined (bound/instantiated/written). Read operations implicitly comprise synchronization (see Section 2.2).

3. **compensate actions.** Transactions allow runtime selection of compensations, i.e., procedures that are to be activated, if the global transaction decides to abort and the completed subtransaction must be semantically undone.

4. **prepared phases** are executed by the commit procedure of a transaction when it is sure that the assignment of communication object values will be possible, i.e., the transaction will succeed. In database terminology, the prepared phase signals other concurrent processes that this transaction commits.

2.2 Synchronization

Synchronization between processes running in parallel is achieved by reading and writing of communication objects. If a communication object is read, and it is still undefined, the read operation must wait until another parallel running process writes this object (cf. logic variables in concurrent logic languages [21]).

3 Prolog&Co and C&Co

This section briefly describes how Prolog and C are extended towards coordination capabilities without destroying existing language constructs. The resulting $^{\mathcal{V}}\mathcal{P}_{\mathcal{L}}$[1] (Vienna Parallel Logic) [18] language is more expressive than FCP(:) [23], one of the most powerful representatives of the family of concurrent logic programming languages (CL) In contrast to CL and other distributed Prologs [14] $^{\mathcal{V}}\mathcal{P}_{\mathcal{L}}$ not only supports atomic unification and communication via shared logic variables, but also provides backtracking and reliability of communication. The notion of communication objects is introduced into C to enforce persistency as well as a rigorous single assignment property for C data types. The semantics of C's operators are extended to deal with communication objects.

3.1 Creation of Communication Objects

The idea of communication objects fits very well to the logic programming paradigm, because in contrast to variables in imperative languages, a variable in logic based languages can be assigned a value only once. Two types of variables are available: *ordinary variables*, corresponding to Prolog's logic variables, and *communication variables*, which have similarities with variables in CL languages. During backward execution, bindings of ordinary logic variables are undone, whereas bindings of communication variables remain forever. $^{\mathcal{V}}\mathcal{P}_{\mathcal{L}}$ variables are ordinary (non-communication) variables by default. Communication

[1] More precisely, we should speak of Prolog&Co, but as we started with $^{\mathcal{V}}\mathcal{P}_{\mathcal{L}}$ we will keep this name for the language.

variables are created explicitly by the prefix operator "#" (see Example 4.1).

Communication objects in C&Co are defined with the `comm` type qualifier (cf. `static`). It can prefix any common C data type declaration, causing the creation of a communication object of the specified type. `comm` automatically converts all members of a complex data type to communication objects. The advantage of this approach is that all C types remain usable, but some of them can be defined to be communication objects. It is also possible to declare members of structures as communication objects. The C&Co system then builds a distributed object tree (see Section 2.1.1) for all members explicitly declared with `comm` (components `left` and `right` in Example 4.2). Now it is possible to work on different members of this structure concurrently by passing shared pointers to different machines. Communication objects in C&Co can also be created dynamically with `new`, e.g. `(comm int cobj).new` creates the communication object `cobj`.

3.2 Concurrency

AND and OR concurrency exist inherently in the logic programming model: like CL languages, VPL provides language constructs to control intra-process parallelism.

VPL clauses have the form "$Head \leftarrow Goal_1 \circ \cdots \circ Goal_n$" where "$\leftarrow$" stands for one of the *rule operators* ":–" (sequential), "::–" (parallel), or "<-" (neutral). All clauses with the same predicate as the head must use the same rule operator. The clauses are tried sequentially, in parallel, or in an implementation defined manner (cf. with the neutral operators in Parlog [13]). "$\circ$" stands for the *commit operator* "|" (see Section 3.3) or one of the *conjunction operators* "&", "&&", or "and". The sequential AND operator "&" is used between goals G_1 and G_2 if execution of G_2 is to be attempted after G_1 has succeeded. The parallel AND operator "&&" is used between goals G_1 and G_2 if execution of G_2 is to be attempted in parallel to execution of G_1. The neutral AND operator "and" is used when the choice between parallel or sequential execution can be left to the compiler or run-time system.

The primitives process(*System*, *Goal*, *PID*) in VPL and `(`*System*`;` *Func*`;` *PID*`).process` in C&Co spawn an independent parallel process at the software system *System* (specified by its InterNet address) executing the goal *Goal* or calling the function *Func* respectively. The communication variable *PID* serves as process identification and eventually becomes bound to the result state of the new process. It can be used to send signals to the process by using the primitives signal(*Signal*, *PID*) in VPL and `(`*PID*`;` *signal*`).signal` in C&Co. *Func* specifies the C&Co function to be executed and its arguments. It is an expression of the form `(`*cobj-list*`).`*funcname*`(`*parameter-type-list*`)`. "`(`*cobj-list*`)`" is called the *interface* of the function. Communication objects which are passed in *Goal* or specified in the interface are shared between the concurrent processes.

3.3 Transactions and Commitment

VPL. The commit operator "|" resembles the sequential conjunction operator, but additionally cuts alternatives and defines the *scope of a transaction*: all goals between two commit operators, or between the head and the first commit operator in a clause, belong to a transaction. Goals to the right of the last commit operator, or goals of clauses without a commit operator, belong to the transaction in which the clause was called. Transactions used in other transactions are named *subtransactions*. The following VPL-predicates [2] are termed *transaction predicates* since they can be used only within transactions: The *full unification operator* "=#="/2 binds communication variables. Initially, the bindings are only visible locally in the scope of the transaction. On transaction commitment they become globally visible. The predicate cvar/1 tests whether its argument is an unbound communication variable and ensures that its argument remains unbound until commitment of the transaction in which it was called. The argument of the predicate prepare/1 defines a predicate which is activated on commitment, if it is sure that all cvar/1 tests and all communication variable bindings, as requested by "=#="/2, can be performed. Several prepare/1 predicates may occur within a transaction — all actions defined by them are called on commitment. If one of these predicates fails, this causes the failure of the transaction. Several compensate/1 predicates may occur within a transaction. The programmer specifies the actions that compensate this transaction. These predicates are collected and activated if the transaction is met during backward execution (*programmed backtracking*). process/3 (see Section 3.2) is also a transaction predicate. All process/3 predicates defined in the scope of the transaction are collected and started upon commitment of the transaction. These semantics have been chosen because otherwise backtracking would cause the loss of process identifiers (PIDs) and processes could no longer be aborted by means of the signal(*Signal*, *PID*) primitive.

[2] The number behind the slash is the number of arguments the predicate takes.

After all goals within a transaction have succeeded all transaction predicates in the transaction's scope are executed atomically. The first "|" in the body of a clause determinates the clause, all other alternatives are pruned away. Not yet tried alternatives in a sequential clause are prevented from execution and the current solution's backtracking information is cleared (removal of choice-points [1]). In a parallel clause concurrently running alternatives are aborted. The control flow of "|" is the same as the COMMIT operator in CL languages, namely a symmetric CUT.

C&Co. A function can be statically defined as a transaction with

```
trans (cobj-list).function(parameters)
  { statements }
(retry; prepare; compensate).commit
```

Statements within transactions can be grouped together with `trans` { *statements* } (*retry*; *prepare*; *compensate*)`.commit`. Every function can be prefixed by an interface and the keyword `trans` to serve as an entry point for external processes. The communication objects in *cobj-list* are passed by reference in contrast to the variables in *parameters* which are passed by value (the usual mechanism in C). `commit` corresponds to the end of the transaction and implicitly performs the *commit procedure*. Transactions are aborted by calling the primitive `abort` which corresponds to a break from a block. All arguments in the interface of `commit` are optional. If the interface is empty, the parenthesis can be omitted entirely. If the transaction cannot commit, the argument *retry* is executed. If it evaluates to `true` (non-zero), then the transaction is retried. If *retry* is omitted, `false` is assumed as the default. A transaction function or a process can be specified as the prepared phase with *prepare* and is called when it is certain that the transaction can commit. If either the transaction or the execution of *prepare* fails, the transaction fails as well. *compensate* specifies a function which is called *after* the transaction has committed and has to be revoked later on, e.g., when another alternative is selected due to function replication or the enclosing transaction fails. Only the compensate action of the outermost transaction is called. It's job is to semantically undo effects of the transaction *and of all sub-transactions*. Thus, compensation does not cascade.

The function `(`*cobj*`).undefined` which can only be used in a transaction, tests if the communication object *cobj* is undefined, and tests whether the object stays undefined also at the point of commitment of the transaction in which it was called. Transaction functions as well as transaction statements can be nested. All statements in between "{" and "}" belong to the scope of the transaction. Global variables and pointer references to local memory are undefined for transaction functions called via `process`. Only the declarations of types and functions, the values of the variables in *parameters*, the communication objects in *cobj-list*, and the locally defined variables and communication objects are known in this scope.

Note that a committed subtransaction's effects become visible globally before the main transaction has committed. This reflects the relaxation of the isolation property of transactions.

3.4 Implicit Synchronization

Synchronization in VPL is achieved by extending the semantics of the unification operator ("=") which is also used for goal-head unification. Goal-head unification, by which clauses in a procedure are selected for execution, can bind ordinary variables to make the goal equal to the head. However, goal-head unification must not bind communication variables. The clause that binds them might not be the one finally selected. Thus, unification is delayed until the needed communication variables are bound by concurrent processes.

The semantics of all operators in C&Co which operate on communication objects must be extended. Synchronization is achieved because the C&Co system has to wait until all objects that have to be read are defined.

3.5 Implementation

We have two VPL prototype implementations under investigation: (1) an implementation of in Prolog which represents a runable formal specification. This prototype system is available and runs on the Internet, using UNIX sockets. (2) An efficient implementation in C based on the Warren Abstract Machine [1], extending it with concurrency and transactions. These prototypes are available. Send Email to the authors for further information.

Communication objects based on C data types are more complicated to maintain than Prolog data types, because they can include (recursive) pointers and compound data types. In VPL it is uniquely decidable when a variable is defined, whereas a complex C structure can be partially defined (i.e., some members of the structure are defined and some are not). In the current implementation of C&Co which is still under investigation we allow incrementally instantiated communication objects.

4 Communication

We present the joint maintenance of a distributed participants list for a conference. Parts of the registration are handled in the USA, Europe, and at a site in Asia. The European and the Asian sites use VPL whereas the USA site uses a C&Co programming system for the administration. The corresponding VPL and C&Co programs are shown in Example 4.1 and Example 4.2.

The participants list is represented as a binary, sorted tree. The Vienna site creates a communication variable (#*Tree*) which is passed by means of the process primitive to the other sites, starting a thread that executes the function `maintain` at these sites, and sharing the communication variable which represents the root of the tree.

Example 4.1 (*VPL Program in Vienna and Tokyo*).

```
% insert new node
insert(P, Tree) <- cvar(Tree) and #Le  and #Re  &
    Tree=#=tree(P, #Le, #Re) | .
% insert into left subtree
insert(part(N1,A,B,C,D,E),
      tree(part(N2,_,_,_,_,_), Le, Re)) <-
    N1 less N2 | insert(part(N1,A,B,C,D,E), Le).
% insert in right subtree
insert(part(N1,A,B,C,D,E),
      tree(part(N2,_,_,_,_,_), Le, Re)) <-
    N1 greater N2 | insert(part(N1,A,B,C,D,E), Re).
% complete incomplete entry
insert(part(N1,A,B,C,D,E),
      tree(part(N2,F,G,H,I,J), Le, Re)) <-
    N1 equal N2 &
    part(N1,A,B,C,D,E)=#=part(N2,F,G,H,I,J) | .

maintain(Tree) <-
    ... & read_participant(P) & insert(P, Tree))
    ...
    maintain(Tree).

start <-
    #Tree &
    process(cco@interbase.cs.purdue.edu,
        (Tree).maintain(), Pid) &&
    process(vpl@tokyo.ac.jp,
        (Tree).maintain(), Pid) |
    maintain(Tree).
```

We have implemented methods so that communication objects can be accessed by VPL and C&Co. Basic data types like integer, float and atoms (character strings) map easily. Additionally we map between structures and terms. The latter ones also have functor name and arity, which optionally can be made visible to the C&Co-programmer. Here it is sufficient to assume that all participating sites know the par-

Example 4.2 (*C&Co Program in Purdue*).

```
struct participant {
    char name[30];          /* participant   */
    Address addr;           /* address       */
    char phone[20];         /* telephone     */
    char email[60];         /* e-mail        */
    int status;             /* member,...    */
    int amount;             /* payment       */
}

struct node {
    struct participant p;
    comm struct node *left;  /* l.  successor */
    comm struct node *right; /* r.  successor */
}

typedef comm struct node CNODE;

trans (CNODE *n).insert(struct participant p) {
     if((n).undefined) {
          (CNODE nn).new;
          n = &nn;
          n→p = p;
     } else if( greater( n->p.name, p.name ) )
               (n->left).insert(p);
          else if( less( n->p.name, p.name ) )
               (n->right).insert(p);
          else      /* incomplete entry */
               (n->p).copy_struct(p);
} (TRUE;;).commit

trans (CNODE *root).maintain(void) {
     ...
     (root).insert( read_participant() );
     ...
} commit
```

ticipants and tree structure. However, as terms can be created dynamically in Prolog based languages, we also support a dynamic way to convert a VPL term into a C structure: the term is accessed by interpreting it according to a description which is passed as part of the communicated object.

All sites may access the tree and insert new participants concurrently. In C&Co as well as in VPL the left and right successors of a node are separate communication objects that can migrate independently of their ancestors.

If a new participant is inserted, only the name must be known at this time. All other fields can be added later, possibly also by another site, as a registration might be submitted to two places[3].

Note that pointers which are communication variables are simply left undefined until they are really used due to the single assignment property. To keep the example simple, we have also omitted the treat-

[3] The C&Co function `copy_struct` only copies fields that are still undefined in `node`.

ment of duplicate names and updates.

The insertion of a new participant is done within a transaction. If the insert procedure finds an undefined node, then it tries to insert the new participant at this place. However it may happen that another site inserts a node there concurrently. Then one of the transactions will fail. If the failing transaction is at the C&Co site, the retry expression, i.e., `TRUE`, will be evaluated, which causes the re-execution of this transaction. Due to the recursive structure of the transaction `insert`, the computation will continue at the father node and thus no computational effort is wasted. The new participant will be inserted in the left or right subtree, and eventually an empty slot will be found in the tree where no other insertion attempt takes place. If the failure occurs at the VPL site, the first clause of the insert procedure cannot commit and eventually another clause will be taken instead, correspondingly causing the insertion in the left or right subtree.

5 Comparison with Linda

Although there are a lot of important programming languages that have similarities with the proposed approach, Linda's orthogonalism was the main motivation to decompose VPL into a coordination and computation part. A comparison between VPL, Orca, and other logic based concurrent languages can be found in [18].

Linda and our "&Co" communication model both offer write once, read many communication objects (in Linda called *tuples*) that processes access in a synchronized way. Communication objects cannot be modified. Communication with infinite data structures is possible: a communication object can reference another one [6]. Black-box communication is possible, meaning that the contents of a communication object can be any kind of data, provided the receiver and the sender agree on its interpretation. With communication objects a high degree of hardware abstraction can be achieved. Processes that communicate via communication objects need not be aware of each other (anonymous communication) and need not even run at the same time. The Linda model can be emulated by any "&Co" language — a Linda implementation in terms of VPL is given in [18]. The main differences between Linda and "&Co" are:

- The addition of "&Co" to a language $\mathcal{L}$ results in an integrated language $\mathcal{L}$&Co, like the here presented C&Co and VPL languages. Our goal is to embed the extensions for coordination as smoothly as possible into the host language $\mathcal{L}$ — only extending it by explaining its behavior if persistence is added to its usual data structures. The programmer has to learn just a few extensions — but may continue thinking in his/her preferred paradigm, without additionally operating in a new sphere, like the tuple space (even if the tuple space is a conceptually easy to understand model).
- Garbage collection is possible in "&Co" - languages, because a process may only access communication objects to which it possesses a reference, which is passed in its interface. Thus it can be determined whether a communication object is still referenced. For the same reason — not all objects are accessible from every process — the name space problem can be avoided in "&Co"-languages. There is no necessity for named communication objects; an internally maintained unique object identifier suffices. More security can be provided because unauthorized processes cannot fetch the communication objects.
- From the beginning, "&Co"-languages have been designed to support transactions and thus reliability. Many communication objects can be written in one atomic step, for example, our Linda emulation allows for several in/out/rd/etc. calls to be grouped into one transaction. Communicated data can be recovered after failures. Recently such extensions to Linda towards persistency have been proposed [11].
- "&Co"-languages have been designed to cross the scope of one computer architecture: whereas typical Linda applications run on one site and all processes are created by one program, a typical "&Co"-application will manage processes running on different sites and accommodate the communication between programming languages that belong to different paradigms.

6 Conclusions

A toolkit has been presented that supports transactions on shared data objects. This toolkit can be used to extend traditional programming languages, designed for single sites, towards coordination. We have shown how Prolog and C can be enhanced by coordination, resulting in the languages Prolog&Co (VPL), and C&Co. A main aspect of these new languages is that the programming style of the host language is not violated, but with a few linguistic extensions a lot of expressiveness and functionality is added.

"&Co"-languages ease the cooperative work between distributed autonomous systems. They are based on an advanced transaction model and support semantic compensation and function replication. Compensation has turned out to be a powerful concept that allows subtransactions to make their effects visible early. Long-living activities, like cooperative processes, decompose into subactivities that have to produce intermediate results, used by subsequent activities. If later on such results become invalid, they cannot simply be withdrawn, but are semantically compensated.

We have discussed how to implement communication objects in a network. In recent work we have specified a language independent coordination kernel [16] which generalizes the ideas presented here and deals with failures, and have shown its application for the coordination of autonomous databases. As a further extension, we have incorporated object-orientation which has not been described in this paper. The recovery not only of communication objects but also of computations in indeterministic programming languages after failures is subject of our future work.

Acknowledgments

We acknowledge the support and the many helpful comments of M. Brockhaus, the head of the Department at the TU Wien, on this paper. Discussions with C. Sabitzer, T. Tschernko, and L. Wei about this work are greatfully acknowledged.

References

[1] H. Aït-Kaci. The WAM: A (real) tutorial. Technical Report PRL research report 5, Digital Paris Research Laboratory, 1990.

[2] H. Bal and S. Tanenbaum. Distributed programming with shared data. *Computer Languages*, 16(2):129–146, 1991.

[3] Ph. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[4] A. Brogi and P. Ciancarini. The concurrent language, Shared Prolog. *ACM Transactions on Programming Languages and Systems*, 13(1):99–123, January 1991.

[5] O. Bukhres, e. Kühn, and F. Puntigam. A language multidatabase system communication protocol. In *Proceedings of the 9th International Conference on Data Engineering.* IEEE Computer Society, April 1993.

[6] N. Carriero and D. Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, 21(3):323–358, September 1989.

[7] Graeme N. Dixon, Graham D. Parrington, Santosh K. Shrivista, and Stuart M. Wheater. The treatment of persistent objects in arunja. In *Proceedings of the Third European Conference on Object-Oriented Programming ECOOP89*, pages 169–189, July 1989.

[8] 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.

[9] A. K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications.* Morgan Kaufmann Publishers, 1992.

[10] H. Garcia-Molina and B. Kogan. Node autonomy in distributed systems. In *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems*, pages 158–166, Austin, Texas, December 1988. IEEE Computer Society Press.

[11] D. Gelernter. Multiple tuple spaces in Linda. In M.R em E. Odjik and J. C. Syre, editors, *PARLE'89*, pages 20–27. Springer Verlag, 1989.

[12] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2), February 1992.

[13] S. Gregory. *Parallel Logic Programming in PARLOG. The Language and its Implementation.* Addison-Wesley, England, 1987.

[14] P. Kacsuk and M. J. Wise, editors. *Distributed Prolog.* John Wiley, 1992.

[15] e. Kühn. Multidatabase language requirements. In *Proceedings of the 3rd International Workshop on Research Interests in Data Engineering, Interoperability in Multidatabase Systems, RIDE-IMS-93.* IEEE Computer Society, 1993.

[16] 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.

[17] e. Kühn, W. Liu, and H. Pohlai. Scheduling transactions on distributed systems with the VPL engine. In *Second Biennial European Joint Conference on Systems Design, ESDA-94*, London, July 1994. ASME. to appear.

[18] e. Kühn, H. Pohlai, and F. Puntigam. Concurrency and backtracking in $\mathrm{V}^{ienna}\mathrm{P}^{arallel}\mathrm{L}_{ogic}$. *Computer Languages*, 19(3), July 1993.

[19] e. Kühn, F. Puntigam, and A. K. Elmagarmid. Multidatabase transaction and query processing in logic. In A. K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 9. Morgan Kaufmann Publishers, 1991.

[20] 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.

[21] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, September 1989.

[22] G. Wiederhold, P. Wegner, and S. Ceri. Toward megaprogramming. *Communications of the ACM*, 35(11), November 1992.

[23] E. Yardeni, S. Kliger, and E. Shapiro. The languages FCP(:) and FCP(:,?). *New Generation Computing*, 7:89–107, 1990.