Implementing Cooperative Software with High-Level Communication Packages

Alexander Forst and eva Kühn
University of Technology Vienna, Institute of Computer Languages
Argentinierstr. 8, 1040 Vienna, Austria, Europe
{alex,eva}@complang.tuwien.ac.at
http://www.complang.tuwien.ac.at/

Abstract

The use of appropriate tools is crucial for the development of robust and distributed software. The programming of heterogeneous environments is more demanding than programming single, stand-alone computers. We believe that client/server technology is not a satisfactory solution. Most problems do not naturally decompose into an asymmetric client/server structure. Better abstraction mechanisms are needed. We propose a new coordination framework that we have developed. It supports shared objects as reliable communication media, advanced transactions, and concurrency through processes that form reliable software contracts.

For a discussion, we compare the realization of a typical distributed application, that belongs to the domain of cooperative work, with three different tools: Our coordination framework, a representative of the classical client/server and message paradigm, and the Linda communication model.

1. Introduction

Distributed and parallel programming [21] is usually considered more difficult than sequential programming. Motivated by our work in the area of heterogeneous transaction processing in multi database systems [3, 15, 16] we have developed a framework, called COKE (Coordination Kernel), for the coordination of autonomous systems. The COKE is an alternative approach for traditional *middleware* like the widely used client/server technology, of which many believe that it is only an intermediate step in the evolution of distributed programming [20]. It uses shared objects (communication objects) that are written within transactions. Every process can rely on communicated data because communication objects can be written only once. Visible data will always be valid. The write-once property has the big advantage of simplifying the implementation of a consistent view on the shared objects in the network. Every written object is reliable and can be recovered after system failures.

This framework can be embedded into different existing programming languages without destroying their basic programming style. This leads to a family of coordination languages (with different programming paradigms like logic, imperative, and functional). Each representative is able to communicate with all other members because everybody uses the same communication objects for data exchange and synchronization.

In this paper we want to show the implementation of a team editor as a simple example of cooperative work. The focus of this work is not on the social issues of computer supported cooperative work but the technical realization.

We compare the implementation that has been carried out with our Prolog based coordination language PROLOG&CO with other communication paradigms, like the message passing one (e.g., with the PVM Parallel Virtual Machine [11]), and with the shared data structures paradigm of the Linda communication model [4, 5]. The criteria for the comparison are communication effort, linguistic elegance, extensibility of the code, and failure behaviour.

In result we find that an advanced coordination paradigm not only satisfies the high linguistic requirements, but provides acceptable performance. Moreover, the same editor implementation can run on parallel hardware without modification, because communication objects provide a uniform mechanism for intra- and inter-process communication. A further distinguishing feature is that a fully recoverable version was easily built by employing the COKE's process mechanism.

In Section 2 we give an overview of our coordination framework, the COKE and the resulting class of coordination languages. In Section 3 we explain the design criteria of the simple team editor. In Section 4 we sketch ad-hoc solutions in PVM, C-Linda, and PROLOG&CO.

2. The Coordination Kernel

The coordination kernel (CoKE [17]) is the heart of our family of coordination languages. It handles communication objects, i.e. it is responsible for the creation of communication objects, the writing of values to them in an atomic step, and their storage on stable media.

The CoKE accepts commands to handle processes. Processes can be started and otherwise manipulated on an arbitrary site. The CoKE is also responsible for the transaction management.

There is only one CoKE process per site. It talks to other processes on the local site and to other CoKE processes anywhere else on the network (see Figure 1 later in this section). There is no global coordinator or master task. For the communication with remote CoKEs UDP sockets are used. There is a fix socket port reserved for the CoKE. For the communication with the local software systems (e.g., with PROLOG&Co) the CoKE uses UNIX pipes.

The transaction model of the CoKE to write communication objects is based on the Flex transaction model that has been developed for multidatabase systems [7]. This model allows an early commit of arbitrarily nested transactions, i.e., it relaxes the isolation property of the classical transaction model [2]. Intermediate results of long running activities can be made visible early. Thus, it is not necessary to hold locks on local software systems over the entire lifetime of a global transaction. This would violate the local system autonomy.

As a consequence, the model includes also semantic compensation of already committed transactions. The programmer specifies compensate actions that semantically undo the effects that have been committed but are not needed because of the failure of a global transaction. In that case, the CoKE automatically starts the appropriate compensate actions.

2.1. Communication Objects

The shared objects of the CoKE are termed *communication objects*. Every process can start another process on the network and use communication objects as parameters for communication and synchronization purposes. Communication objects have a *single assignment* property and thus can be written only once. Their values are immediately stored on stable storage and can be recovered after system crashes. As a major consequence, every process can rely on communicated data. There is no need for cascading aborts of computations, because visible data will always be valid. The write-once property has the advantage of simplifying the implementation of a consistent view on the shared communication objects in the network. The programmer has the vision of an arbitrarily shared memory where she can start processes anywhere, communicate data back and forth, perform synchronized computations, but without the additional effort to handle network and system failures because the CoKE relieves her from these tasks.

Communication objects are written only within transactions. At the point of commit, all values are made visible to other processes in one atomic step. From this point, the values of the involved communication objects can never be changed again. Every process can rely on them. Even more, if a site is unreachable because of network failures or down because of a system failure, the committed value is automatically transmitted to the site when it can be reached again.

The transactions are built and behave according to the Flex transaction model [7], i.e., the programmer can build arbitrarily nested transactions. She can also define compensate actions that are automatically executed by the CoKE in order to undo the effects of nested transactions whenever an enclosing transaction must abort.

Processes that want to read a communication object which has no value yet (i.e., communication objects which are *undefined*) must wait until another process writes to this communication object. An arbitrary number of processes can be synchronized that way.

2.2. Processes

The CoKE provides means to spawn processes on remote sites or on the local site. The type of the process can either be *independent* or *dependent*. There are some subtle differences between processes of these two types.

Independent Processes. An independent process is a reliable, autonomous software contract with the CoKE, that assures that the process will be executed, even if system failures occur. The CoKE recovers all independent processes that have not yet terminated, and spawns them again—with their old image and the same arguments as before—after a failure or shutdown. As communication objects can be passed in the argument list of the process, the restarted process will automatically obtain access to the same communication objects it was allowed to see before. Communication objects are persistent and survive failures. Thus, a process can proceed its work with the same communication objects and can see all committed values.

Independent processes play an essential role for recoverable programming, as they define checkpoints for recoverable computations.

Dependent Processes. A Dependent process is started from within a transaction to perform part of the transaction on another site. It is not automatically recovered after a failure. The CoKE tries to start it as long as there is no system failure at its site. After the process has been started successfully, the CoKe does not care for it any longer. But in contrast to independent processes, the transaction from which the dependent process was called depends on the successful completion of the process. The transaction commit must wait until all dependent processes that were called in its scope have successfully terminated. Thus, the transaction and all its dependent processes have the same point of commit.

2.3. Languages

Existing programming languages (or any system, e.g., a database system) can use the functionality of the CoKe either by calling library functions or we can embed the framework smoothly into the languages without destroying their basic programming style and conception. A language $\mathcal{L}$ which follows the first approach is termed $\mathcal{L}$&Co ($\mathcal{L}$ + coordination). A language with embedded coordination features is termed Co-$\mathcal{L}$ (Coordination Language $\mathcal{L}$). This leads to a family of coordination languages of different programming language paradigms, like logic, imperative, and functional.

So far, we have embedded the toolkit into imperative and logic based programming languages. Prolog&Co extends Prolog by library functions, whereas VPL (Vienna Parallel Logic[1]) [18, 10] extends Prolog using the second approach. The C extension C&Co uses library functions and Co-C [8, 9, 10] also follows the second approach.

Every programming language which has been extended in either of the two ways communicates with the coordination kernel which runs on the same site. The CoKe handles every communication object which is passed to it and communicates with other CoKes at other sites (see Figure 1).

Each representative is able to communicate with all other members of the "&Co-family" via shared communication objects, provided the corresponding semantic language translations are foreseen. The CoKe can translate most data structures automatically (e.g., Prolog terms to C structures). Other structures must be converted manually. For example, if you use a communication object which represents a list (from a fictional Co-Lisp) in an imperative language like C&Co you have to convert the list into a corresponding C structure for yourself.

The concept of communication objects makes all &Co-languages interoperable. It aims at a re-use of existing program and data resources guided by the principles of coordination. This way, software systems can remain small. Instead of incorporating all features into one software package, services are "bought" at other systems [23]. Note that there is no difference in the programming of the *buyer* and the *provider* of the service in contrast to the *client* and the *server* of the classical client/server model.

[1] The name should be Co-Prolog but for historical reasons we keep VPL.

3. The Design of the Team Editor

We want to discuss the implementation of an editor that supports a cooperative editing process of an arbitrary number of users in a network of distributed UNIX workstations. At any time all users must have the same view of the joint document. To simplify the task, only a line oriented team editor (TeDi) shall be built with a set of simple rules solving the "social" aspects of competing for the same text paragraphs. The focus is solely the realization of the communication and synchronization problems and not a new, sophisticated semantics for cooperative work. The criteria for a comparison are: Communication effort, linguistic elegance, extensibility of the code, and failure behavior.

The basic functionality that shall be supported by this simple TeDi are the insertion and deletion of lines and the substitution of words within a line. Concurrent insertions referring to the same line shall be resolved by allowing the insertion of *all* lines—in an arbitrary order—*before* the specified line. If two users want to substitute the same word in the same line, only one shall be allowed to do so. If there exist several instances of the word in this very line, one user may change the first occurrence and the other one the second. The concurrent deletion of the same line shall be resolved by granting the right to delete the line to the "faster" user process and causing the failure of the other user's deletion command. A substitution/deletion conflict of the same line shall imply that either one must fail, and analogous for a deletion/insertion conflict.

A print command serves to refresh the view of the text; it provides the user with the up-to-date view of the text.

The joining of a new user at a specifyable site and display shall be supported to allow more than one user. The addition and removal of new users shall be dynamically scalable; this is of particular importance, if we assume that the text editing is a long lasting action during which the number of participating users may change.

Then, we want to enhance this simple team editor by the following additional features and evaluate the communication effort and the linguistic elegance.

Fault-Tolerance: The edited text must not be lost if failures occur. Moreover, every client process shall automatically be recovered so that the client can continue its work as if no failure had occurred.

Controlling the Commit Granularity: We want to discuss how the simple editing commands like insert, delete, and substitute, that refer to one line only, can be grouped to commands that modify a couple of lines in an atomic step. For example, we want to consider the necessary changes to support (α) the insertion of an entire paragraph, consisting of several lines, (β) the deletion of several lines at once, or (γ) the constraining of

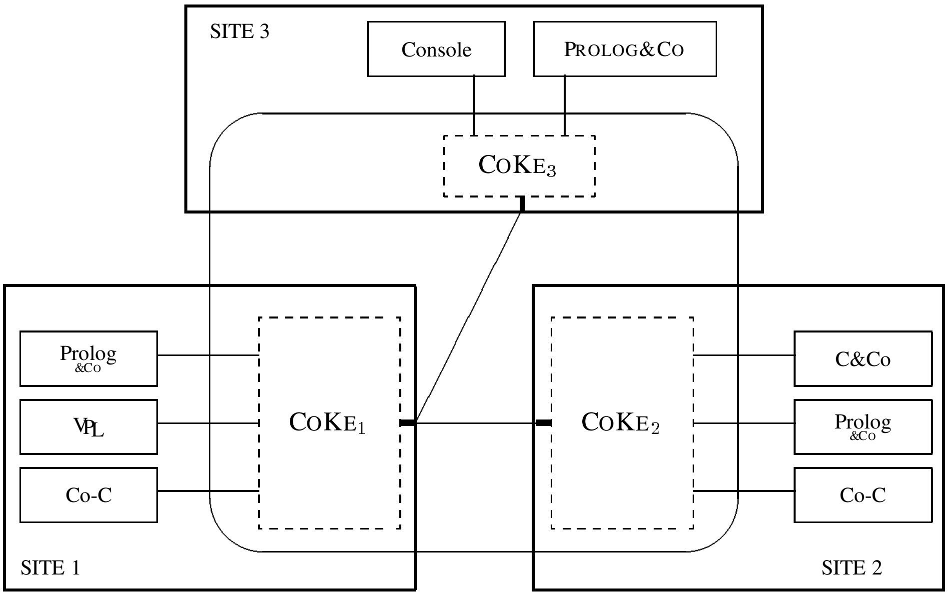

Figure 1. Architecture of the Coordination Framework

an insertion depending on the fact that the previous line ist still existing (i.e., not deleted by another user in the meantime).

4. Implementing the Team Editor

We assume a tree as the common data structure to represent the entire text in order to give a reasonable framework for a comparison. Each text node consists of three parts: The line of text it stands for, and two pointers to the next text nodes (left and right, respectively; cf. Table 1).

pointer to left text	line representation	pointer to right text

Table 1. Text Representation

This structure is better suited for the dynamic insertion and deletion of text lines than a linear linked list. The granularity for changes is—without restriction of generality—limited to single words. The flattened tree represents the current view of the entire text in the correct order.

We ignore efficient allocation and removal of dynamic storage. Deleted nodes may simply be marked as deleted and remain in the tree structure.

A unique line identification cannot be achieved simply by ascending line numbers. Assume for example that the users Bill and Jane concurrently attempt to insert lines; if Bill who wants to insert a line at the beginning of the text is faster, he will cause Jane's view of the line numbers—Jane wants to insert at the end of the text— to be incorrect. Therefore, let us use unique but not ordered line identifiers (`LID`s) to tackle this problem. Let us also assume—without restriction of generality—that a `LID` is constructed from the unique name of the user who has created the line plus a local counter to ease the extension of the editor to cope with partial failures.

The particular solutions in PVM, C-Linda, and PROLOG&CO presented in the following sections are "ad-hoc", that means we tried to use the corresponding development tools in an as natural as possible way to avoid artificial performance hacking.

4.1. A PVM Solution

PVM [12] is a C library that makes message handling simpler than using sockets. It supports a set of blocking and non-blocking point-to-point and broad-cast sending and receiving routines, the possibility to define dynamic process groups, and barrier synchronization. PVM runs on a variety of parallel and distributed hardware architectures. Transparency of the different architectures is achieved by using XDR. Internal send and receive buffers are used to pack and unpack the communicated data. In summary, PVM follows the message passing principles so let us therefore map the TEDI task into a client/server structure.

One process is designated as the text coordinator who is responsible for the mutual exclusive access of conflicting operations. A new user can join the cooperative editing process by requesting the basic text operations at the server. The joining requires that the server has a unique name (e.g., using a special process group) that the client processes are aware of and to which they may send their insert, delete, substitute and print requests. So for the PVM solution, two programs must be written: The TEDI-server and the TEDI-client.

Unfortunately the server process forms a bottleneck for all text accesses. If the server process fails or the server is separated by a network partition, all clients must suspend their work. Moreover, every time a print command is issued, the entire text—which can be very large—is packed into a message buffer and communicated to the requesting client.

The former problem cannot be overcome easily. The only possibility would be to replicate the server process which will require complicated synchronization between all server processes to guarantee mutual exclusive text access. Many programs which are based on the client/server architecture suffer the same scalability problem and try to solve it with varying success [14].

The latter problem can be solved by asking all clients to administrate a local copy of the text. All requests are still sent to the server which decides whether the request can be performed or not. Requests that caused text changes are then broadcast to all client processes to issue the same changes on their local copies. This is well supported by PVM, because message ordering is guaranteed. However, the problem that we faced with this improvement was that now the client continuously has to wait for two events: update messages from the server and input messages from the user. For this, no real language construct, comparable to UNIX's `select` function exists in PVM.

The reason seems to be that here the two worlds of PVM's communication facilities and the UNIX i/o collide. A possible solution is—consistent with the PVM model—to split each client into two processes: the real client plus a keyboard process. The keyboard process can be spawned by the client in another X-window and can then communicate with its corresponding client.

The next problem into which we ran with this design was that PVM's input/output handling—which is quite crucial for an interactive system like TEDI—did not allow an easy solution to enter and display text in the keyboard window; we ended with entering text in the keyboard process and displaying text in the real client window. A sound solution seems to require even more complicated structures.

4.1.1 Fault-Tolerance

In the PVM program, a client may fail at any time without compromising the other participants nor destroying the text's integrity. This is because clients do not administrate data.

However, the main problem is the server bottleneck. The server is solely responsible for the maintenance of the text. All clients depend on it. A server crash causes the entire text—that clients already may have seen—to be lost. To tackle this problem, the server must store every text change reliably on a secondary storage medium, before allowing clients to see it. More precisely, the server should use a data file and log file mechanism, like a database system, to guarantee that at any time either one of the two files is correct.

4.1.2 Controlling the Commit Granularity

A straight forward extension for PVM would be to introduce a new message for every required feature (like (α), (β), (γ)). This increases the different types of messages to be handled by the server. For a dynamic and more flexible tuning of the granularity, a new protocol must be designed that provides messages for the marking of the start and end of actions, and the specification of a set of operations that belong to one action. It is the server's task to collect all operations of one action and to perform then all or none, if the end marking message is sent. More or less, one must implement a transaction manager for the server. Note, that you have to change both programs: the server as well as the client to handle the new protocol.

4.1.3 Communication Effort

The communication effort is small in the client/server solution, but the answers to print requests will become uncontrollable large. The schema is to send one request to the server, which performs the entire task without issuing further communication. Clearly the server is a bottleneck in this architecture, as no client can work autonomously on parts of the text but some of these problems can be overcome by caching strategies.

4.1.4 Linguistic Elegance

This solution follows the classical client/server principles. In particular, for problems that concentrate on the sharing of large amounts of commonly used data, this is not a suitable approach. When the clients compete to change the shared data this is even worse. The decomposition into a server (which has led to a severe performance bottleneck) turned out to be quite unnatural.

4.2. A C-Linda Solution

C-Linda is the embedding of the Linda model into the C programming language [22]. Linda is a communication model based on shared data structures, called *tuples*, that are located in a global pool (termed *tuple space*). Concurrent processes can jointly access the shared tuples by means of the four atomic operations `out`, `rd`, `in`, and `eval` that write and read tuples to and from the tuple space.

`out` writes a tuple in the tuple space. It never blocks.

`rd` reads one tuple from the tuple space. The returned tuple must match with the argument of `rd`. Thus, the parameter serves as a template to specify the desired tuple. `rd` blocks until a matching tuple is found in the tuple space. If more than one tuple matches, one is returned non-deterministically. If you prefix an argument with "?" then its value is read from the tuple space.

`in` behaves like `rd` but it removes the found tuple from the tuple space.

`eval` creates a process tuple, that is a tuple that is computed by a newly spawned, concurrent process. After the computation completes, the tuple is put into the tuple space and evaluates to the return value of the process. It can be accessed like any other data tuple.

This small number of access functions that allows the easy specifications of all possible kinds of communication structures [4] determines the elegance of the Linda model.

For the following discussion we will refer only to the original Linda model[2], without taking into account recent proposals to extend Linda (e.g., guards, atomicity and fault-tolerance [1], or structuring of the name space [13]).

A text node is represented as a tuple `("text", lid, line,` $\texttt{flag}_{deleted}$, $\texttt{lid}_{left}$, $\texttt{lid}_{right}$`)`. The tuple `("root", id)` points to the root of the text.

If two users want to insert text at the same point at the same time, the following will happen. The users Bill and Jane want to insert text before the same line and, for example, Jane's process is faster in performing the `in` operation that temporarily removes a text node from the tuple space. That causes Bill's process to block in its `in` operation until Jane writes back the modified text node into the tuple space. Now Bill's process is able to proceed. It will find the new line, descend the tree appropriately, and, finally, insert Bill's new text line also before the original line.

C-Linda's possibility to spawn a new process with `eval` is used to add a new user to the system. In contrast to the PVM solution, a new user can only be invited by another user to join the session.

4.2.1 Fault-Tolerance

Each tuple space operation is atomic. Thus, the failure of a user process cannot hamper the global consistency of the text. Other users are not disturbed by such a failure. Only the site where the text is maintained is crucial. If—comparable to the server process in the PVM solution—the site that is responsible for the maintenance of the tuple space fails, the entire text is lost. Thus, it is mandatory to store every change of the tuple space on secondary storage.

[2] We have used the C-Linda version for the Sequent Balance [22].

4.2.2 Controlling the Commit Granularity

The granularity control is not a severe intervention to the Linda solution.

For (α) we can think of a solution that inserts all lines that form a paragraph into the tuple space at first. After that, the last line that links the paragraph to the rest of the text is added. However, we have to take care not to violate consistency, if we require that either all or none of the paragraph's lines are inserted. We also must take into account that the user process may crash while inserting the lines of text. Unfortunately, only the C-Linda operations *per se* are atomic actions. It is not possible to group operations to larger actions in the original Linda model. Thus, the programmer must employ a transaction model to take care of that.

(β) and (γ) could be solved by removing the tuples which represent the lines which should be deleted temporarily from the tuple space, checking whether the operations can be done, and finally returning the lines. Here, we also have problems with the global consistency that require more sophisticated algorithms if we assume failures. Another problem is that other users cannot access the lines of text that we have removed temporarily to perform the necessary operations. This decreases the possible degree of parallelism.

4.2.3 Communication Effort

Let us consider every tuple space operation as one communication step. For the insertion of lines, communication is required, to (1) locate the required line before which the new line is to be inserted, and (2) recursively communication is needed to descend in the text tree to find the right place where to insert the new line. Moreover, two communication steps have to be added for incrementing the global counter. Deletion of a line only requires two communication steps: the removal of the line and then the insertion of it into the tuple space.

Most communication effort arises for the print command: to print the entire text, every line requires one communication step.

4.2.4 Linguistic Elegance

The TEDI program in C-Linda was developed most quickly of all three solutions. The text structure could be mapped easily into appropriate tuples. Direct access of tuples is possible, because tuples have global names and thus allow an associative access—on the other hand one is confronted with the known disadvantages of a global name space.

4.3. A Prolog&Co Solution

Communication objects are write-once; thus a line of text in PROLOG&CO is represented as an infinite stream where the last entry is the current version of the line. A text node is therefore the list `[LID,LineStream,DeletedFlag,Left,Right]`, with `LineStream` being a stream `[Line|Lines]`.

The implementation of the basic TEDI functions in terms of PROLOG&CO is as follows: To start the cooperative editing process, first a new communication object (`Text`) must be created and then the first user process must be spawned as an "INDEP" process, executing the goal `tedi(Text,UserName)` which starts the editor.

Every user can dynamically join another user by issuing a `join/3` command. This causes a new TEDI running in another X-window to pop up at the new user's site and display.

Like in the C-Linda example, during insertion of new text lines, the program descends the tree and tries to write the corresponding communication object in a transaction. If two users concurrently attempt to insert a text node at the same place, one of the two corresponding transactions will not be able to commit. This will cause backtracking and the choice of another alternative for insert, which in turn will continue to descend the tree recursively until the line of text can be inserted. The transaction eventually will succeed because we cannot assume that there will always be a competing user asking for insertion of text at the same place.

A line is valid if the flag `DeletedFlag` is an undefined communication object. Analogous, if `Left` resp. `Right` is an undefined communication object, this indicates the end of the tree at this point.

4.3.1 Fault-Tolerance

The presented PROLOG&CO program is already reliable. All text lines are represented as communication objects that are written in transactions and are thus automatically recovered after failures. The joining of new users is done via processes of type 'INDEP' so that they are also automatically recovered by the COKEs. The only small extension was to add a stream, representing the current line counter, to the `tedi/2` goal. Thus, after a restart also the line counter stream is recovered. Let the last entry in the stream stand for the currently highest value used. Then, after a crash, the user will continue to number his lines with unique identifiers.

4.3.2 Controlling the Commit Granularity

The granularity of text changes is reflected by the use of transactions. In the above solution, every basic operation (insert, delete) immediately commits its effects. If the commit is delegated to its caller (the user), the caller can issue an arbitrary number of basic operations and commit them together. This immediately solves (α) and (β). For (γ) it is possible to ensure that `DeletedFlag` of the referred line is still undefined at the point of commit of the transaction.

4.3.3 Communication Effort

To give precise figures about the communication effort here, one must know the details what messages are exchanged by the communication protocols [19] between the COKEs. We will give a brief estimation for the default protocol (based on passive replication with a primary copy migration schema).

No communication is needed to find the right place to insert a new line of text because every COKE maintains a copy of all text nodes. However, after a line has been inserted, every participating site is automatically informed about the new line which requires $2N$ messages (one for communicating the new line and one for acknowledgement) if N is the number of different sites. For deletion, the location of the line also can be found in the locally stored communication objects without additional communication effort. Similar to above, $2N$ messages are sent to inform all COKEs about the new value of `DeletedFlag`. A major advantage is that viewing the whole text also requires no further communication effort because every COKE already possesses a replica of every communication object it may see. The entire text is automatically replicated on all COKEs where TEDI processes run.

4.3.4 Linguistic Elegance

The distinguishing property of the simple prototype is that it already provided fault-tolerance and that the code does not contain low level networking commands.

5. Conclusions

Every one of the three languages turned out to be suitable to build a quick prototype. However, many of the capabilities that are supported by the COKE (and thus by every &CO-language) must be handled by the application programmer in PVM and C-Linda. These include the implementation of replication strategies, transaction management, and logfile based state storing techniques to support recovery.

We believe that if a tool burdens too much low-level details on the programmer, the development of robust, distributed applications and their maintenance will become uncomparable more difficult than the development of sequential applications. The motto must be to support distributed application programming as *specifying* the task at hand instead of *hacking* it.

Prototypes of the CoKE PROLOG&Co and C&Co are available for a number of UNIX workstations. Send E-mail to `eva@complang.tuwien.ac.at`. A full-screen variant of the simple TEDI using Motif has already been implemented by using the C&Co language [6].

6. Acknowledgements

The support and encouragement of Manfred Brockhaus is acknowledged. Many thanks to Herbert Pohlai for his many helpful comments on this text; to Minh Dang for writing a first prototype of the team editor in PROLOG&Co, and to Christian Wehrl for the discussions about the C-Linda and PVM solutions.

References

[1] D. E. Bakken. *Supporting Fault-Tolerant Parallel Programming in LINDA*. PhD thesis, University of Arizona, Department of Computer Science, August 1994. TR 94-23.

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

[3] O. Bukhres and e. Kühn. Highly available and reliable communication protocols for heterogeneous systems. *Information Sciences*, 3(1):1–40, January 1995.

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

[5] N. Carriero and D. Gelernter. Coordination languages and their significance. *cacm*, 2(35):96–107, 1992.

[6] H. M. Dang. TEDI, ein Editor für kooperatives Arbeiten im Netzwerk. Master's thesis, University of Technology, March 1996. In german.

[7] 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.

[8] A. Forst. Implementation of the coordination language C&Co. Master's thesis, University of Technology, September 1995.

[9] A. Forst, e. Kühn, and O. Bukhres. General purpose work flow languages. *International Journal on Parallel and Distributed Databases*, 3(2), April 1995. Special Issue on Software Support for Work Flow Management, Kluwer Adademic Publishers.

[10] 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.

[11] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine*. The MIT Press, 1994.

[12] 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.

[13] 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.

[14] A. L. Ibbetson, P. F. Linington, I. A. Penny, A. B. Smith, and G. E. W. Tripp. Reducing the cost of Remote Procedure Call. In A. Schill, C. Mittasch, O. Spaniol, and C. Popien, editors, *Distributed Platforms*. Chapman & Hall, London, UK, 1996.

[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. *Principles of Coordination Systems – A Library Approach for Distributed and Reliable progrsamming in C and Prolog*. TU Vienna, 1996. The CoKe Reference Manual for Version V.0.8.2.

[18] 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.

[19] e. Kühn, H. Pohlai, and F. Puntigam. Communication and transactions in $\mathrm{V}^{ienna}\mathrm{P}^{arallel}_{\mathrm{L}_{ogic}}$. *Computers and Artificial Intelligence*, 13(4), 1994.

[20] T. Lewis. Where is client/server software headed? *IEEE Computer*, 28(4), April 1995.

[21] T. Lewis and H. El-Rewini. *Introduction to Parallel Computing*. Prentice Hall, 1992.

[22] A. Osterhaug. *Guide to Parallel Programming On Sequent Computer Systems*. Prentice Hall, 1989.

[23] C. Pancake. Software support for parallel computing: Where are we headed? *cacm*, 34(11):52–64, November 1991.