flow systems.

The isolation concept used in ConTracts is a powerful means to express the isolation needs of workflow processes. It was built to allow processes to release access restrictions on shared objects as soon as possible without violating their correctness. Therefore a new correctness criterion has been introduced which allows parallelism far beyond serializability. Furthermore, the design allows us to handle conflicts explicitly instead of reacting to a conflict by simply aborting the current transaction.

Although the prototypical implementation of a ConTract system (APRICOTS) [Schw93] has been under development for more than three years now, not all features have been completely implemented yet. We are currently working on the efficient implementation of the concurrency control concepts. Additionally, we are looking at the problem on how existing transactional environments like e.g. Encina from Transarc can be enhanced to support transactional workflows like ConTracts directly. Likewise, we are looking at database systems and other "resource managers" to find out on how they can be leveraged to join a ConTract environment.

References

[Elma92] Ahmed K. Elmagarmid (ed.), Database Transaction Models for Advanced Applications, Morgan Kaufmann Pub., 1992

[DE95] U. Dayal, M. Hsu, C. Mohan, M. Rusinkiewicz, F. Schwenkreis, Panel Session: Workflow Automation, International Conference on Data Engineering, Taipei Taiwan, 1995

[GrRe93] J. Gray and A. Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann, San Mateo, Calif. 1993

[RIDBS94] M. Stonebraker (ed.), Readings in Database Systems 2nd edition, Morgan Kaufman Pub., 1994

[SaLo89] B. Salzberg, D. Lomet, Access methods for multiversion data, Proc. of the ACM Intern. Conf. on Management of Data SIGMOD, 1989

[Schw93] F. Schwenkreis, APRICOTS- A Prototype Implementation of a ConTract System: Management of the Control Flow and the Communication System, Proc. of the 12th Symposium on Reliable Distributed Systems, 1993

[Schw94] F. Schwenkreis, A Formal Approach to Synchronize Long-Lived Computations, Proc. of the 5th Australasian Conference on Information Systems, 1994

[Stone87] M. Stonebraker, The Design of the Postgres System, Proc. of the 13th Intern. Conf. on Very Large Data Bases, 1987

[Svob81] L. Svobodova, A Reliable Object-Oriented Data Repository for a Distributed Computer System, Proc. of the 8th Symp. on Operating Systems Principles, 1981

[WaRe92] Helmut Wächter, Andreas Reuter: The ConTract Model, chap. 7 in [Elma92]

state of concurrently accessed objects which in result guarantees the executability of the *Compensation-Step*. The constraints defined after *Steps* are called *Invariants*.

In detail, the mechanism works as follows:

- The programmer assigns predicates with *Steps* in the *Script* which represent the *Invariants*
- At execution time the run-time system checks at the end of the execution of a *Step* if the predicates are valid. If they are valid, the constraint is established and the transaction which protects the *Step* is allowed to commit.
- If the constraint cannot be established the ConTract-Manager is informed. That means the ConTract-Manger gets information about an isolation conflict and is able to initiate actions to resolve the conflict.

As introduced in the original model, *Invariants* can also be used to support the executability of a ConTract. For example, a *Script* can contain two *Steps*: checking a flight and the reservation of a flight. Instead of grouping those two *Steps* under one transaction and restricting the access on the flight database more than needed, a predicate can be established after the first *Step* which expresses the pre-reservation of a certain amount of flight seats. This ensures the executability of the second step without restricting the accessibility of other seats. On the other hand, this not a real *Invariant* of the whole ConTract because the constraint can be removed after the second *Step* has committed. Hence, another "class" of *Invariants* has been defined which supports the executability of a ConTract instead of ensuring correctness.

Both ways to use *Invariants*, to ensure correctness and to support the executability provide a powerful mechanism to express the isolation requirements of a process. Furthermore, they allow us to release unnecessary access restrictions as soon as possible. For example, it was found out that there are many cases where only the existence of an object has to be guaranteed, while other arbitrary accesses can be allowed.

Another important feature of our isolation concept is the explicit handling of problems caused by a conflict on shared resources. There is built-in direct support for defining special parts in the *Script* which should be executed if the evaluation of a predicate fails. Additionally, in case of a conflict with an *Invariant* defined to support the executability of another process, negotiations can be supported (e.g. to grant access temporarily).

4 Summary and prospect

Two remarkable features supporting workflow in the ConTracts project have been presented in this paper: advanced data models and advanced concurrency control mechanisms.

The data model of ConTracts supports the usage of high-level constructs and the examination of a process' history as requested by users. It is not sufficient to provide them only with transactional storage. Since the data objects used by workflow processes are getting more and more complex (e.g. multi-media documents), the evolution of such an object is very important and the information about which activity has modified the object in which way must be provided by work-

same name the semantics are unique because of the assigned *parallel index* which can be used from now on for recovery or history examination.

The addressing schema of *context-elements* is also be used on the programmers level, i.e. that programmers assigns *context-elements* to the parameters of *Steps* like e.g. parameter1 = *context element*. Several defaults are provided to make the programming of the *data flow* easier:

- The current parallel index is set automatically and can be used by a *special name*.
- The current loop-counter can be used by a *special name*.
- Other *Steps* can be referenced by using their identification (also supported by the graphical user interface).
- If only the variable name is given the newest version is used.

3.2 Invariant based synchronization

As said before, serializability is not a good correctness criterion for long-running processes like workflow. Hence, the correctness of ConTracts had to be newly defined [Schw94]. The first step to define a new correctness criterion was to investigate the users' view of correctness which resulted in the observation that there are at least three states of a process which are correct:

- The state when a process has never been started: *initial state*
- A state which is defined equivalent to the *initial state*: *compensated state*
- The state of the process when it is successfully executed: *final state*

Since it is necessary to avoid total isolation in workflow processes, preliminary results will become visible outside of an executing process. Hence, the transactional approach of atomicity (all or nothing) cannot be realized because a *rollback* mechanism requires the isolation property. Therfore, other mechanisms have to be used to "undo" the work done by a process if it is necessary to cancel the execution of a process: we call these mechanisms *compensation*. To *compensate* in the case of ConTracts means that for every committed *Step* a so-called *Compensation-Step* has to be executed which represents the (semantical) counter-action of the original *Step*. If all necessary *Compensation-Steps* have been executed successfully the *compensated state* is reached.

A correctness criterion for a ConTract can then be defined as follows:

The execution state of a ConTract is correct iff:

1. The ConTract is in an *initial state* or
2. The ConTract is in a *final state* or
3. If neither 1 nor 2, then for every committed *Step* the execution of its *Compensation-Step* is ensured.

To ensure the correctness of ConTracts according to the definition above, it is assumed that the prerequisites to execute a *Compensation-Step* are known when the corresponding *Step* has been executed. This means that after the execution of a *Step*, constraints can be established to ensure a

element. Instead, a new *version* is generated which represents the actual value of the variable. This approach can also be found in database systems supporting the so-called *time domain addressing* [Svob81][Stone87][SaLo89]. Since the examination of the history of a workflow process is usually done based on the process definition and not on a time basis, the access of the different versions of *context elements* on a time basis is not very helpful. Furthermore, the support of high-level programming constructs cannot be solved by a time-based access mechanism.

During the investigation of the requirements it was found out that the different ways of addressing *context elements* are independent from each other. From an abstract point of view this means that a multi-dimensional addressing schema is needed. Since the *time domain addressing* approach represents only a two-dimensional approach (identification and time) it is not sufficient.

Our investigation of high-level programming constructs has discovered that the following dimensions are needed:

- *Sequential Steps*: New variable versions are created by sequential execution of *Steps*.
- *Loops*: Another dimension can be found when loops are executed.
- *Parallel Branches*: The third identified dimension considers the case of parallel branches generated by the run-time system to realize e.g. a *parforeach* statement.

Since we store the *context* in a transactional storage which is used by several ConTracts two more dimensions are needed to address a *context element*: the identification of the running ConTract instance and the name of the *context element*.

The five dimensions introduced above are the basis for the addressing schema which is used to store and access elements of the *context*. That means to access a *context element*, a 5-tuple (variable-name, contract-id, step-id, loop-counter, parallel index) in which we assume that the contract-id identifies the ConTract instance in the system and the step-id identifies the creating *Step-instance* as defined in the *Script* of a ConTract.

Since the last four elements can be seen as the unique identification of the step-instance which created the version of the *context element*, we can compress the schema to (variable-name, step-instance-id). However, the user will the access the context indirectly providing the same information.

A short example will show the usefulness of the introduced concept:

- A programmer defines (implicitly) a parallel branch in a *Script*:
  `parforeach person of S step_send_call_for_papers(receiver = person)`
  where person is a string variable and S is a set of strings defined in the *context*.
- At execution time a so-called *Pseudo-Step* is executed which generates as many versions of the *context-element* (person) as there are elements in S. These versions can be identified by the *parallel index*.
- Then as many versions of the *Step* as elements in S are generated which will be visualized on the user interface. They are also identified by the *parallel index*.
- Even if all of the visualized *Steps* have the same name and all of the input variables have the

solved by a proper data model (support of high-level programming).

- Especially in case of parallel executions it is very important to have the capability to define an explicit data flow or "data piping" from one *Step* to another *Step*. That means that a proper data model has to provide an explicit addressing method which can be used at the user level (explicit addressing).

2.2 Isolation requirements

Similar to the requirements for the data model, the long duration of workflow processes causes special requirements for isolation. The classical transactional approach of establishing locks on touched objects is inapplicable for processes which are possibly running for months. Furthermore, it is in most cases unacceptable to abort or simply delay an execution if a concurrency problem occurs. Considering applications of the real world, the following requirements can be identified:

- Total isolation is neither required nor meaningful for a workflow process.
- Real applications need a flexible mechanism to express their isolation needs properly.
- Conflicts caused by isolation problems (in most cases) cannot be solved by an automated system mechanism. Instead, explicit conflict handling by the application itself has to be supported.
- Due to the fact that so-called "groupware" systems should be integrated with workflow systems concurrency control mechanisms for workflow systems must be powerful enough to support cooperative work.
- As a result of the points above, the correctness criterion for workflow processes cannot be serializability. Instead, a new criterion has to be defined on which proper mechanisms can be built.

3 Ideas and approaches

Motivated by the requirements introduced in the section before, a lot of work has been done to define a proper data model and powerful concurrency control mechanisms for the ConTract model and its prototypical implementation APRICOTS. A short overview of the concepts is presented.

3.1 Context - the data model of ConTracts

As introduced in [WaRe92] the definition of a *Script* contains the declaration of "ConTract-local" variables - *context elements* which built up the *context*. These variables are kept in stable transactional storage and are only accessible from the ConTract in which they are defined.

One of the major characteristics of *context elements* is the fact that their value can only be affected by *Steps*. This means that only *Steps* of a ConTract can read or modify the value of *context elements*. Another important feature is the transactional protection of *context elements*. Since the *Steps* are running under the protection of a transaction, their modifications of the *context* are protected by this transaction. This results in the fact the effect that changes to *context elements* can always be associated with committed *Step-instances*.

To support the examination of the history, *context elements* are stored in an update free way, i.e. every modification of a *context element* does not overwrite the old value of the *context*

different styles of talking about consistency, for example.

Starting from these considerations, we have developed a programming model called ConTracts [WaRe92] and a prototype run-time system to go with it, called APRICOTS. The functionality provided covers the whole range of requirements that are meanwhile discussed under the heading of “workflow management”.

2 Critical points

In this article, we will focus on the concepts which are not yet addressed by other approaches.

First the usage of (nested) transactions in ConTracts has a direct impact on the following discussion it is necessary to say a few words on this topic. It is an important feature of ConTracts that every *Step* of an execution is running under the protection of a classical transaction and that several of these *Steps* can be grouped to run inside the same top-level transaction. Additionally, there is a guaranteed forward recovery which ensures that once started, a ConTract never gets lost or stuck in an inconsistent state. Instead, it will either be executed to its “normal” end or reset to an acceptable state by a compensating mechanism (if requested by the user).

2.1 Requirements for the data model

The data model for workflow systems has not yet been discussed in the literature, though it is an accepted fact that there are special requirements for a data model which supports long-running processes [DE95]. We believe the following the following are among the requirements for workflow systems:

- Users are very interested in examining the history of workflows. That means that they want to be able to get information of executed processes regarding e.g. information about “which path was taken why?” or “who has done what?” (support of history keeping and examination).
- Since *forward* recovery is a very needed feature in workflow systems the information needed to continue an execution after a failure must be easily accessible and as persistent and recoverable as the process itself (recovery support).
- At the definition or programming level, nobody wants to use the “low-level” notation of the (workflow) engine. Instead, a high-level programming mechanism is wanted to be used (either graphical or textual) which has to offer well-known programming constructs like e.g. loops, branches etc. Most of these classical programming expressions are using “program variables”, i.e. loop counters or variables to define predicates of branches, which have to be easily depictable on the data model too.

  Another problem of using “high-level” programming constructs appears in combination with the need to support the history-keeping of a process (either for examination by the user or for a compensating mechanism). Since the user usually defines a single variable to be used e.g. as a loop counter which will be overwritten every time the loop is executed, the system is unable to examine the variable’s state at an arbitrary point of the execution. This problem can also be

In: Bulletin of the Technical Committee on Data Engineering Vol.18 No.1, IEEE Computer Society

ConTracts - A Low-Level Mechanism for Building General-Purpose Workflow Management-Systems

Andreas Reuter *Friedemann Schwenkreis*
Institute of Parallel and Distributed High-Performance Systems (IPVR)
University of Stuttgart
Breitwiesenstr. 20-22
D-70565 Stuttgart, Germany
email: {Andreas.Reuter, Schwenkreis}@informatik.uni-stuttgart.de

1 Introduction

The ConTracts project was started to investigate the possibilities of handling long-lived computations in a database context. Note that the emphasis on databases is not a restriction. It simply acknowledges the fact that all long-lived computations - by their very nature - will accumulate a certain amount of state, will access shared data and will interact with the environment and with other long-lived computations. In other words, they perform a large number of activities that are properly supported by database systems. Hence, the discussion of long-lived computations and the definition of the role of databases in that context cannot be separated.

At the time the project started, there were some approaches we could refer to. The first line of suggestions tried to generalize classical ACID transactions by adding more features, giving them more flexibility, etc. Proposals of this sort are sagas, flex transactions, split transactions, to name a few [Elma92][RIDBS94]. Another group of proposals took a much more pragmatic perspective by defining extra functions in existing TP-monitors that allow to chain two subsequent transaction programs, for example. Some database systems support chain transactions, there are mini batches [GrRe93], and a number of other ideas to try and work around the problem.

Our approach, called ConTracts, which will be described in this paper, is characterized by three very simple insights:

- Extending the transaction model does not solve the problem, because a long-lived computation is much more complex than an ACID transaction. On the other hand, nobody is helped by a type of transaction, which is neither A, nor C, nor I, nor D.
- Managing long-lived computations requires more systematic approaches than pragmatic extensions to existing operating systems and TP-monitors.
- Apart from its close relationship to databases, handling long-lived computations is a problem of proper language abstractions. Whereas simple modules (and transaction programs, for that matter) have a system residence time of 10^{-1} sec, a long-lived computation can be around for a number of years; i.e. something in the order of 10^{8} sec. Quite obviously, phenomena which are 9 orders of magnitude away from simple modules, need different programming techniques and