in Proc. of the 12th Symposium on Reliable Distributed Systems, Princeton (NJ), 1993; IEEE Computer Society Press

APRICOTS – A Prototype Implementation of a ConTract System
– Management of the Control Flow and the Communication System –

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

Abstract

The principle of transactions has been proven in the field of database systems. However, there are many fields where classical transactions are not suitable to model the actions of the 'real world'. An approach to extend the principle of transactions is the ConTract model, which weakens some demands of the ACID-principle, but at the same time includes other features. In this article, we introduce an approach to implement a ConTract system. Especially, we dwell on the implementation issues of a component for the reliable control flow management and of a transaction-oriented communication system.

All methods discussed in this article are developed at the IPVR in a project sponsored by the DFG (German Research Society) called: 'Objektbanken für Experten' (Objectbases for Experts)[1] .

1 Introduction

Classical transactions are only suitable in a limited way for the modelling of long-lived actions [6]. A projection of distributed actions on this model is also hardly possible. Because of the need to run long-lived actions reliably, that means with a well defined fault-semantics, in a heterogeneous environment, the transactional concept has been modified and extended in several approaches [4]. One approach is the ConTract-model [15], which includes many strategies and possibilities, especially in the fields of fault-tolerance and fault-handling.

The concepts to run ConTracts were developed within a project of the DFG (German Research Society) at the IPVR and are implemented in a prototype system (APRICOTS – **A Pr**ototype **I**mplementation of a **Co**n**T**ract **S**ystem).

The following sections give an overview of the actual work. Section 2 is a short introduction to the ConTract model. In section 3, the architecture of the system is presented. Section 4 deals with the transaction-oriented communication system. The control flow management is presented in section 5. A summary of the implementation and the future work for the system is found in section 6.

2 ConTracts

ConTracts are suitable for the modelling of long-lived actions on the basis of ACID-transactions. For a ConTract it is guaranteed, that the execution is done in a finite amount of time either successfully, or the original state of the ConTract, or a logical equivalent state, is reconstructed via compensating actions. That means, that a ConTract similar to an ACID-transaction accepts only the original state (or logical equivalent) or the state after the successful execution as a final state [15].

This guarantees that in (almost) any cases of failures the ConTract run-time system is able to recover and continue the execution of a ConTract with a minimal loss of work. A good example is the update of bank accounts. If you have to update 1000 accounts and an error occurs after the 950th update action, you do not want to loose the work done so far. It would be acceptable to rollback the 951th update action and continue the work. This is the way a ConTract system handles errors.

2.1 Description of the control flow

The ConTract-model includes a multi-layer structure for the programming model. The control flow is described independently of the execution-units (*Steps* see below) in a so-called *Skript*.

The *Skript* defines the structure of the control flow, using elements for loops and conditional branches as well as elements to define parallel actions. Furthermore, the transaction boundaries are defined on this level, that means the *Skript* defines which elementary actions should run under the protection of a transaction. The system generates a transaction automatically in case there is no transaction definition for an elementary action.

It is easy to recognize that ConTracts are built of several classical ACID-transactions in which the (intermediate-) results are visible. For the synchronization with other Con-

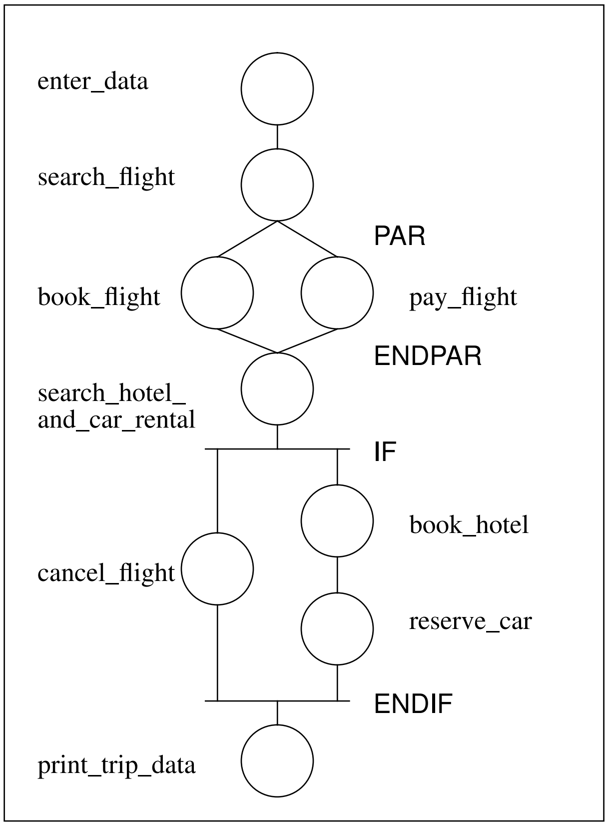

Figure 1: A workflow-example

Tracts running in parallel, there is a possibility to define so-called *Invariant*s. With the *Invariant*s you are able to protect (global) data from the concurrent access of other ConTracts. The *Skript*-programmer defines the strategy how to manage these predicates (e.g. locking, escrow, check/revalidate).

One mechanism for the fault-tolerance you can see at *Skript*-level is connected with the *Invariant*s. The *Skript*-programmer is able to define so-called *conflict resolutions* which are elementary actions executed to resolve the (synchronization-)conflict if an *Invariant* fails.

Apart from the control flow the *Skript* defines the data dependencies between elementary actions. That means, it is defined in the *Skript* which *Step* generates data and which *Step* uses data from another *Step*. This is done in a so-called *Context Declaration*. The *Context* is the (data-) interface between *Steps* and the same reliability as for the control flow is guaranteed for the data in the *Context* [2].

2.2 Executing units

The elementary actions for the *Skript* are the *Steps*. *Steps* are execution sequences which are running under transactional protection. *Steps* unite related actions and offer a possibility for abstracted modular programming without the knowledge of transaction boundaries.

2.3 An example

A good example for a long-lived action executed in a distributed environment is the booking of a business trip. In figure 1, a graphical description of the control flow while booking the trip is given. It gives a short overview of a *Skript* and *Steps*. A detailed description can be found in section 5.

3 The architecture of APRICOTS

APRICOTS is designed strictly modular. The functionally related parts are combined in components and are developed independently of other components. An overview of the components of APRICOTS is given in figure 2.

The design of the system is developed in order to simplify the extension with other components (e.g. with a Log-Manager).

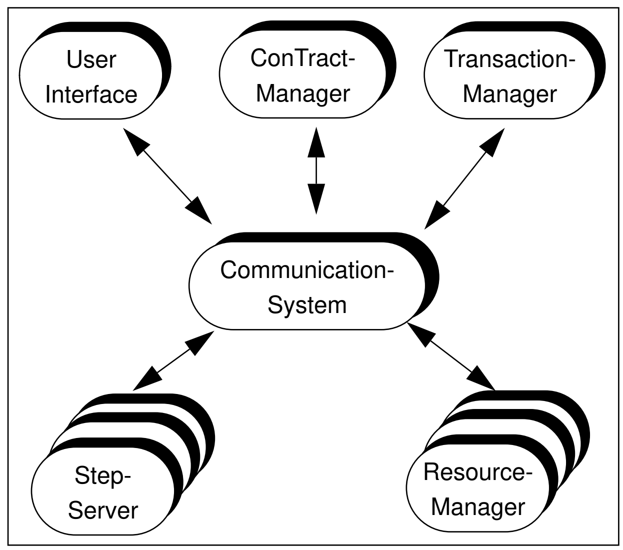

Figure 2: The architecture

3.1 .The ConTract-Manager

The ConTract-Manager has to execute the *Skript*. It has to guarantee the reliable execution of a started ConTract and is responsible for the forward recovery after a crash and for the request for transactions.

3.2 The Transaction-Manager

This component runs the protocol to implement the transactional semantics (e.g. 2 Phase-Commit). We are planning to use the Encina-product from Transarc [11] in our implementation.

For the communication with other components in the system which do some work under the protection of a transaction, the Transaction-Manager can use a standard interface like the XA-interface from X/Open [10].

3.3 The Communication System

The communication system is the central message interface of APRICOTS. It guarantees that every component in the system can send a RPC (remote procedure call) to another component independent of the location – we call it a “location transparent” call. Furthermore, the communication system runs a protocol with the transaction manager to support distributed transactions and extends the RPC to a TRPC - a transaction-oriented remote procedure call.

3.4 Step-Server

The Step-Server is a user-programmed component. It includes the code for *Steps* (s. a.) which can be any user-programmed sequentiell code plus the usage of routines from the communication system. The execution of the *Step*-code is done asynchronously to the execution of the *Skript.*

3.5 Resource-Manager

A Resource-Manager manages an object. It supports functions on an object like a database, a machine etc. Unlike a Step-Server the Resource-Manager is called synchronously. Furthermore, the call is initiated by a Step-Server (or another Resource-Manager) and not by the ConTract-Manager.

3.6 The User Interface

The user interface allows the user to start a ConTract and to manipulate it by using commands. The interface visualizes the *Skript* in a graphical form and allows text-oriented output for every component in the system (monitoring).

4 The Communication System

This section is a short introduction to the concepts of the communication system of APRICOTS. We only present the important features which are needed to support the reliable execution of long-lived applications.

4.1 The requirements

A main issue of APRICOTS is to tolerate component failures at run-time. Furthermore, a flexible reaction on changing load situations should be supported. To satisfy this requirement we need the possibility to change the system configuration at run-time either manually by a system administrator or automatically by the system itself.

To support the flexibility in our implementation, we included some features in a special form of a RPC (remote procedure call) called “RM_Call” [9]:

- The RM_call has an uniform interface (‘normal’ RPCs have an interface similar to a ‘normal’ procedure call).
- We support synchronous and asynchronous calls with the RM_call (RM_call_sync/RM_call_async).
- In a RM_call a ‘service’ is called. The binding of a physical address is done at run-time not at link-time.
- A flexible grouping of services to a server is supported.
- Because the address-binding is done at run-time the server configuration can be changed at run-time (adding a server / removing a server ...).
- The Multiple instantiation of the same service is possible, and therefore, a load balancing can be done.

4.2 Messages in APRICOTS

The communication system is responsible for the correct resolution of the physical address. It is invisible for the programmer where his call will be executed at run-time. To optimize the message-sending the communication system has three different methods to deliver a message (see figure 3):

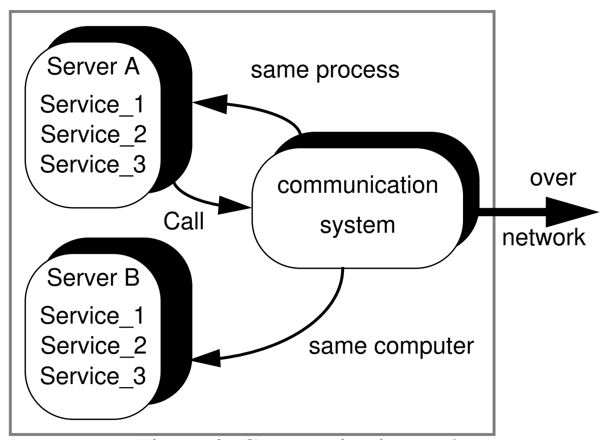

Figure 3: Communication paths

- If the addressed service is in the same process, a ‘normal’ procedure call is executed.
- If the service is on the same computer, the message is sent via message-queues of the POSIX standard [8].
- If the service is found on another machine, the message is delivered via TCP/IP and the local communication mechanism.

4.3 The Service-Call – an abstraction

As mentioned above, you do not have to address a server in a RM-Call, if you want to call a service. From the caller’s point of view, the address mechanism of the system looks like this (see figure 4):

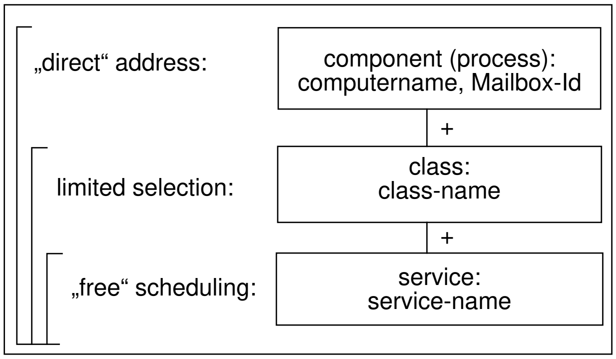

Figure 4: Adressing schema

- There are services in the system which are identified by their name. You use the service-name in the RM-Call to call a service. If there is more than one service with this name and no more information is specified, the communication system schedules the call.
  example: booking_a_flight
- Every service in the system is associated to a class. If a service is addressed by its name and the class-name, the scheduling is limited.
  example: booking_a_flight, Lufthansa
- Once a service has been called, the logical address of the component, including the service, is returned to the caller. This address can be used in future calls.
  example: booking_a_flight, Lufthansa, airport Stuttgart (and more)

4.4 The „postman”

If the service addressed by a RM-Call is not found on the same computer, communication over a network is needed. We use TCP/IP as the protocol for the network communication. Because the components of APRICOTS receive messages over a stable message area (see below), there is a need to transform the network messages. This functionality is done by the so-called postman which is a process to connect TCP/IP with the local communication mechanism. That means, the delivery of messages is done on two levels:

- A component sends messages over computer-boarders to the postman of the computer on which the called component is running (TCP/IP).
- The postman delivers the message to the component that has been called (persistent message queues – POSIX see below).
- The postman can use mechanisms of the communication system to start components if they are not running any longer and to inform the caller about the failure of a component.

4.5 Stable message-areas

As mentioned above, every component should support the crash-security and fault-tolerance of a ConTract. This includes the tolerance of component failures and means for the communication system that a delivered message must not get lost because of a failure. Instead, it is handled by a newly started component.

We use an abstract mailbox system to implement these characteristics. A logical number is used to address the message-area of a component at the interface to the communication system.

We implement the storage-areas of the abstract mailboxes as stable storage-areas generated by the operating system: On VMS and UNIX-systems we use so-called persistent message queues from the POSIX level 4 (draft) standard [8]. They are only associated with a component so that the failure of a component does not destroy the area. Only the failure of the computer (operating system) causes the loss of the data.

4.6 Extension to a TRPC

The mechanisms shown above are the basis for the communication. The communication system of APRICOTS also has to support the processing of distributed transactions. Therefore, we have to implement an interface to the transaction manager.

You need closed nested transactions to implement the ConTract model. The so-called top-level transactions are managed by the ConTract-Manager (BOT, EOT, ...) and all servers which performed some work under the protection of a (top level) transaction have to run a joined commit.

Because the addressing of the servers is done by the communication system, it is necessary to send the address information to the transaction manager. With the addressing information the transaction manager is able to run a 2-Phase-Commit protocol.

The component which is supposed to run the distributed transactions (Encina) has an interface for communication services (in TRAN [5]) to support this functionality, but a part of the transaction-oriented management has to be linked to every component. In APRICOTS we avoid this restriction, because we want to use resource-managers on platforms on which Encina is not available (e.g. on GUARDIAN the operating system of Tandem computers).

4.7 Flexible Server Construction

The flexible construction of servers, and a free grouping of services to run in a process, is one of the main requirements of the communication system. The construction of a component includes two main steps:

- programming of the services as C-functions
- construction of the servers with these functions

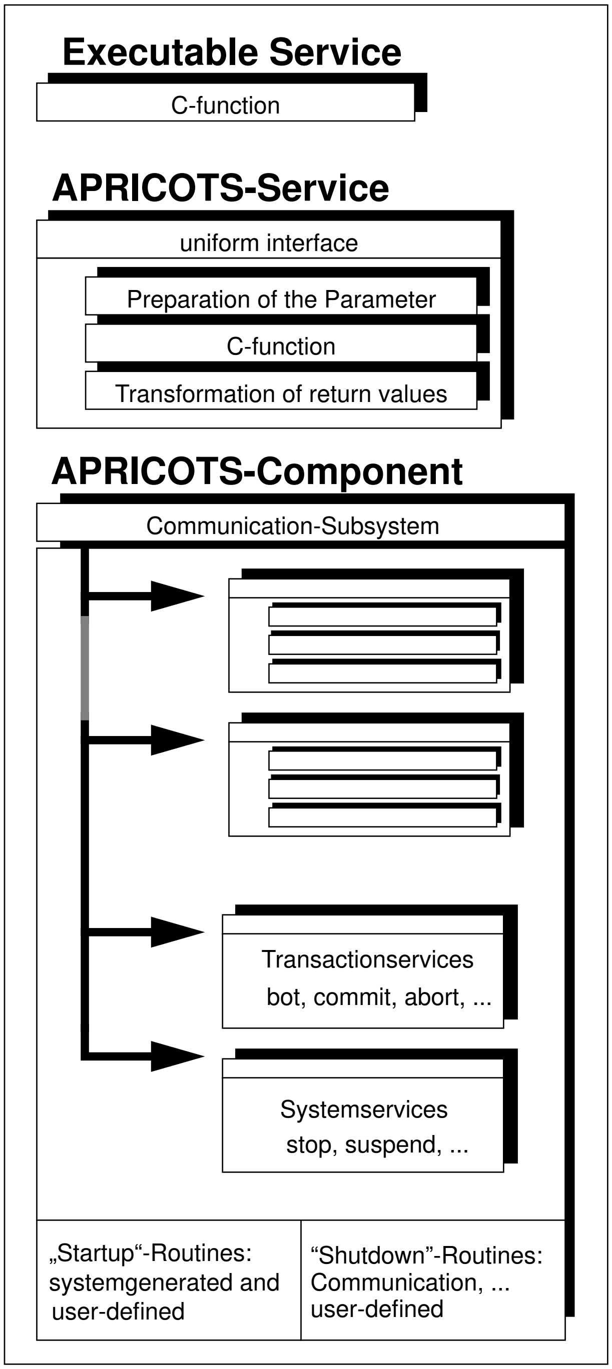

Figure 5: From a service to a server

The programmed functions need an uniform interface to the communication system to support a standard calling-interface in APRICOTS. This interface is generated by a *stub-compiler* similar to the classical RPC. The services can then be included in a component (see figure 5).

In order to guarantee a proper initial state of the component and the possibility for automated recovery, it is possible to include user-defined routines for initialisation (as well as for the release of resources during the 'shutdown' of a component).

Because the components are running in a system for distributed transactions, they have to support the functionality to run transactions with an external coordinator. The components support this with transaction-oriented services which are made available for the transaction-manager by an interface (e.g. XA from X/Open [10]). The real executing and service specific parts have to be programmed by the programmer of the services.

The construction process is supported by a configuration tool [16] which supports the construction of a component with a graphical interface, runs the stub-compiler and starts the component automatically (not the whole functionality is implemented yet).

4.8 Conduct in case of errors

As the communication system resolves the physical addresses at run-time we have the following methods to handle errors:

- mark the lost component as unavailable
- try to reschedule the call (new binding of a physical address)
- if there is no service left, try to start a component (including the required service) automatically with the option to execute recovery-actions for the old component.

5 The ConTract Manager

The implementation of the ConTract manager functions are based on a suitable representation of the *Skript*. In APRICOTS, the *Skript* is defined by a *Skript*-programmer using a block-oriented language[15]. The language is suitable for the definition of a *Skript*, but it is inefficient to interpret it at run time.

5.1 Used Skript Representation

For the efficient execution of a *Skript* it is necessary that the component for the execution control has fast access to the information about the successor *Steps* of a running *Step*. The easiest way to support this is to store this information directly instead of extracting it from a programming code. Therefore, a compiler was designed [12] which transforms the code of the *Skript* in a representation of a *predicate transition net* and stores this representation in a database to support the fast access.

A predicate transition net is a variant of a *Petri net*. Unlike Petri nets the connections from places to transitions are provided by predicates. Because of the predicates a token is sent only from a place to a transition if the predicate is satisfied (see figure 6).

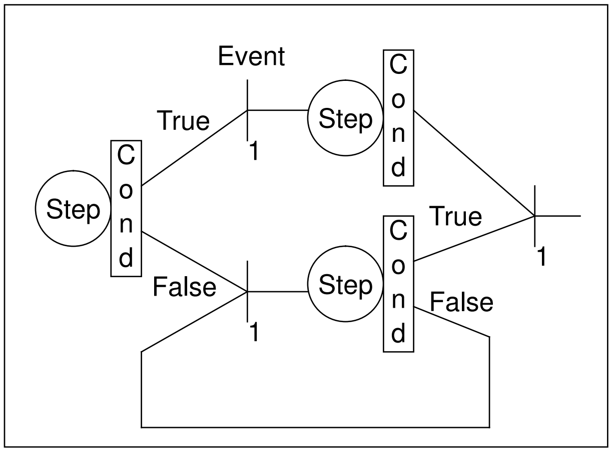

Figure 6: A predicate transition net

In APRICOTS, we use different terms for the elements of a predicate transition net, because the *Skript*-language is not as powerful as the common predicate transition net:

- *Step*/*Pseudostep* for place
- *Event* for Transition
- *Condition* for predicate
- *Trigger* for sending a token

You can see that the terms are similar to the principle of the 'triggers and transactions' [14].

5.2 Processable constructions

The protoype implementation limits the set of processable constructions. But even this subset shows the possibilities of ConTracts. In the following, the constructions with their graphical representation are presented. *Steps* which are marked with capital letters are so-called Pseudo-Steps (PS) which are used to mark special parts of the control flow.

5.2.1 ConTract

APRICOTS is able to execute *Skripts* of the form shown in figure 7. It is easy to recognize that only the „normal" case is supported right now, i.e. that the execution of the compensation *Steps* is not included.

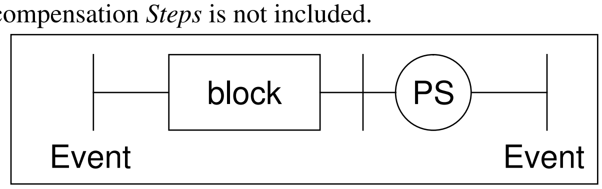

Figure 7: The ConTract

The following constructions are presenting the possible replacements for a „block".

5.2.2 Simple Sequence

The simple sequence is either a single Step or the execution of two blocks one after the other:

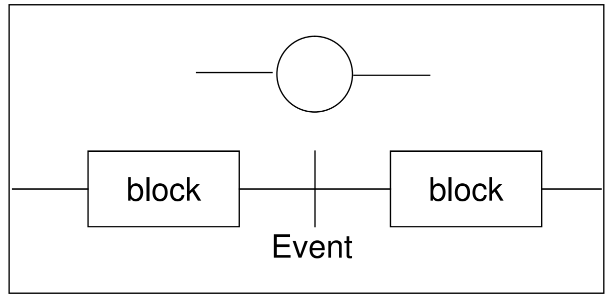

Figure 8: The simple sequence

5.2.3 Parallel-Block.

A parallel-block (figure 9) supports the execution of any number of blocks in parallel, i.e. that they are started at the same time.

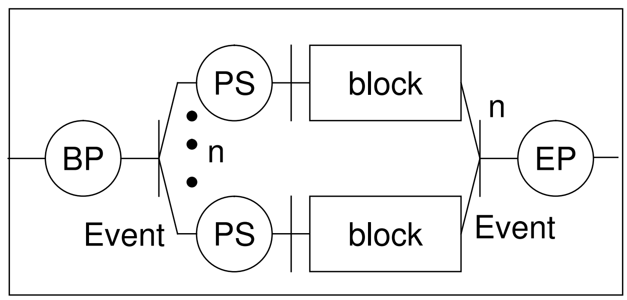

Figure 9: The parallel-block

5.2.4 Branch-block

The structure of a branch-block (figure 10) is almost identical to the structure of the parallel-block, however only one internal block is activated. As an extension to this construction we plan to include the possibility to activate more than one block parallelly.

In the current implementation we support the selection of the block which has to be activated by a predicate which has to be defined as an equality-relation over the Context.

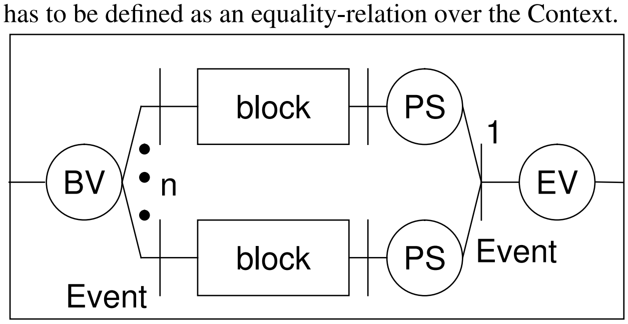

Figure 10: The branch-block

5.2.5 Loop block

We consider „for" and „while" statements to be loop blocks (figure 11). The differentiation is done in the Conditionstep (CS). The usable predicates of the while-statement are the same as the predicates of the branch block.

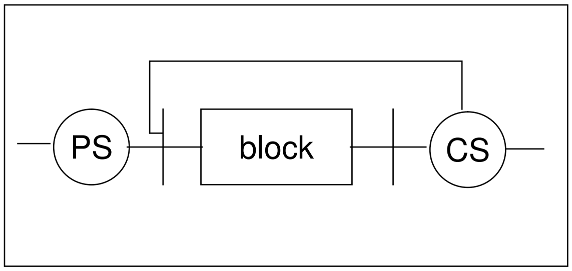

Figure 11: The loop-block

5.2.6 Transaction boundaries

The ConTract model supports the flexible grouping from *Steps* to transactions. This global mechanism cannot be supported in APRICOTS. To group *Steps* which do not belong to the same block in a transaction can be in contrast with the definition of the control flow (because of the ACID properties).

Therefore, we restrict the definition of transactional boundaries to the boundaries of blocks. That means, that the beginning of a transaction (bot) has to fit with the beginning of a block and just as well the end of a transaction (eot) with the end of a block (Pseudosteps are not mentioned).

5.3 Functions of the ConTract-Manager

At the external interface of the ConTract-Manager only the functions which manipulate a ConTract are visible. These functions are introduced in the following in alphabetical order.

5.3.1 *activate* ConTract

This function starts a ConTract. The only parameter is the name of the ConTract that will be started. The function reads the *Skript*, transactional information and dependencies from the database and then activates the first *Step*.

5.3.2 *compensate* ConTract

The *compensate* function activates the compensation of finished and committed Steps. Active or prepared transactions will be aborted.

5.3.3 *migrate* ConTract

With the *migrate* function it is possible to transfer the execution control of a ConTract to another ConTract manager, which is the same as suspending a ConTract and resuming it from a different ConTract manager (e.g. on different computer).

5.3.4 *restart* ConTract

This function runs a recovery of a ConTract. It is needed after a process or machine failure. Active or prepared transactions will be aborted and restarted. The ConTract will then resume its execution – *forward recovery.*

5.3.5 *resume* ConTract

The counter action to suspend (see below), the interrupted execution of a ConTract can be resumed.

5.3.6 *step_finished* receive-function

We will see in the following that the calls of *Steps* are done asynchronously. Therefore, it is necessary to have a special function which receives the return-messages of *Steps* and executes the *Skript* depending on the return values.

5.3.7 *suspend* ConTract

This function makes it possible to interrupt the execution of a ConTract. After the interruption the previous execution of the ConTract is stored on a stable storage – the failure of a computer or process has no influence on a later resume of the execution.

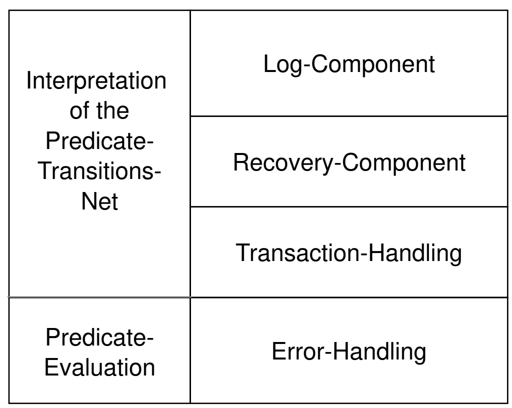

Figure 12: The internal structure of the ConTract-Manager

5.4 The Structure of the ConTract-Manager

The ConTract-Manager has several functional parts (see figure 12). These parts are implemented as abstract datatypes and follow the modular approach of the development of the whole system.

5.4.1 The *Skript* Interpreter

The *Skript* interpreter manages the control flow. Therefore, it reads the translated form of the *Skript* from a database. After the evaluation of predicates (dashed line in figure 12) it decides to activate which *Steps* and then sends an asynchronous call to the services which implement these steps. The external function *step_finished* (see above) is given as the *callback service* which receives the return values of the executed *Step* and continues interpreting the *Skript*.

5.4.2 Error-Handling

Many strategies can be included in the error-handling-part of the ConTract manager. Currently, we only deal with the failure of the evaluation of an *Invariant*. If the *Invariant* fails, a so-called *conflict resolution* is executed which tries to resolve the conflict.

We plan to include more strategies which depend on the return value of a *Step* as:

- limited retry of a call
- migration to other components
- temporary interruption of the execution
- automatic start of components

5.4.3 Logging

The log component records the execution of the *Skript*. All important data for a possible recovery is recorded by this part of the ConTract manager. Furthermore, the log component supports a function to generate stable points, so-called *checkpoints*. In the case of crash a *checkpoint* guarantees a state of the execution which is able to run a forward recovery. That means, if a failure occurs and the restart function is activated, the maximal loss of work is the work done after the checkpoint has been taken.

5.4.4 Recovery

As mentioned above, one of the goals of the ConTract model is to rollback only the minimal amount of work in case of a failure and to continue the execution. This forward recovery is realized by the recovery component. It requests the rollback of all non-committed Steps (at the time the error occured) and resumes the execution. The information needed for the actions is read out of the log written by the log component.

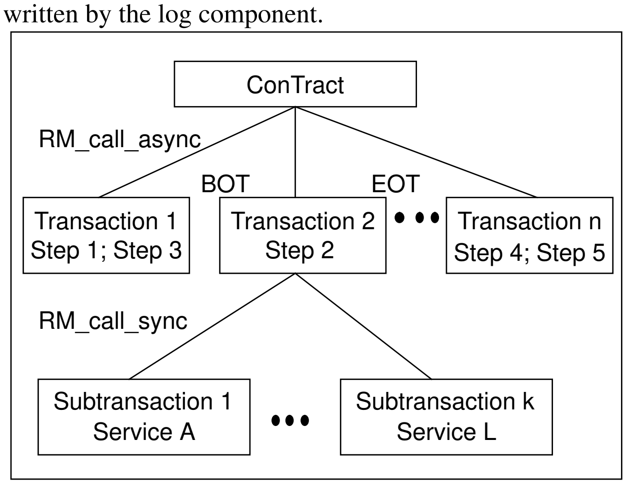

Figure 13: Transaction boundaries

5.4.5 Transaction-Handling

In the description of the communication system we already said that the ConTract manager needs a transaction handling component, which has an interface to the transaction manager. Using this interface the ConTract manager is able to inform the transaction manager about the beginning and the end of transactions and invokes the distributed commit protocol. Furthermore, the component demands the aborting of transactions during a recovery.

For the flexible combination of *Steps* to a transaction it is necessary to make the data, changed by one *Step* under the same transaction, visible for other involved *Steps*. As a consequence the resource-managers have to be able to grant other resource-manager their locks (see e.g.[13]).

5.4.6 Predicate Evaluation

As mentioned in section 5.4.1 the predicate-evaluation-part is needed to decide the correct execution path of the *Skript*.

It is also responsible for the evaluation of *Invariants*, but there is currently no decision which predicates will be supported and by which elements they can be defined.

5.5 Implementation issues

In the subsections before we introduced many features of the ConTract-Manager – the central component to execute the *Skript*. Most of the features have been implemented yet, but without taking perfomance aspects into account.

The whole implementation is done in C using embedded SQL to have access to a database system (Ingres on the UNIX-platform, rdb on the VMS-cluster). This was necessary because we do not have access to a distributed log facility (or repository) like it is included in Encina and we were not able to implement this on our own (less manpower).

However, we implemented the log of the ConTract-Manager as a database relation and gained the very comfortable access-features of SQL. Even the storage of the Skript (after compilation of the language-oriented description) is done in the SQL database to have a stable storage which can be accessed from several nodes in our network.

As soon as we are able to get Encina from Transarc we will port the log-service and the *Skript*-storage part on the distributed log and the transaction-oriented file-system.

5.6 What is implemented yet?

We have already implemented the *Skript*-execution part of the ConTract-Manager. Because the predicate evaluation is not fully developed we restrict the predicates to equality expressions over *Context*-variables and constants.

We support the C-types “long” and “double” and the type “string” as types for *Context*-variables. The access of *Steps* to non-committed *Context*-variables from another *Step* running under the same transaction is also supported.

The reliability features of the ConTract-Manager are fully implemented, that means we are able to recover from a system crash, we have the limited retry of failed *Step*-calls and we execute the Conflict-Resolution in case of a (simulated) synchronization conflict.

Furthermore, we are able to suspend the execution of a ConTract and continue the execution later (on the same computer or a different machine).

All these features show that we also have implemented our communication system.

5.7 What will be implemented soon?

Soon we will add the transactional part of the ConTract-Manager to run the distributed 2-Phase-Commit protocol with all involved components. Furthermore, we will have an example of a Step-Server which includes the fully transactional semantics (including XA) based on the distributed log facility of Encina.

6 Summary and Prospect

The components in APRICOTS are developed to guarantee a maximum of fault-tolerance. The communication system supports this approach with a high level of abstraction in the call-mechanism and with almost stable message-areas. The ConTract manager which is responsible for the robust control flow, realizes this by the logging of the execution.

We have to emphasize that for both components the mechanisms for fault-tolerance and fail-safeness can only be guaranteed if the resource managers are running under the protection of classical transactions. Only in this case it is possible to abort the work of a resource manager or to rollback the work without unwelcome effects.

There are many problems in implementing the transactional semantics, because present database systems do not offer a fully implemented XA interface and are not able to grant locks in the required way. Furthermore, the implementation of a transaction manager which considers the transaction dependencies of the Skript is difficult. Therefore, we develop the transaction manager in cooperation with another project at the IPVR [7].

The last point we want to mention is the synchronization component of ConTracts. This includes several problems which cannot be solved in the scope of the project of the DFG. In future implementations of APRICOTS (in other projects), we will research into this area as well.

We developed user interfaces [3] for using and monitoring the system or are working on the design of them. Especially for the monitoring and management of the *Skript* it is essential to interact by using a graphical interface, because a text representation of a predicate transition net cannot be comprehended.

We will develop a graphical editor which replaces the block-oriented language and the compiler.

The components described above are developed in a UNIX environment. We will soon connect a VMS-Cluster to show the functionality in a heterogeneous environment. The next step will be the connection of a Tandem-Computer to the ConTract-System.

7 Literature

[1] R. Bayer, T. Härder, P. Lockemann (Hrsg.). *Objektbanken für Experten*. Springer Verlag, 1992.

[2] Holger Burkert, Bernd Raichle. Ein Daten- und Verarbeitungsmodell für den Kontext. Doppelstudienarbeit 859, IPVR Universität Stuttgart, 1990.

[3] Marcus Eitler. Eine graphische Oberfläche für die interaktive Steuerung langlebiger verteilter Abläufe. Diplomarbeit 956, IPVR Universität Stuttgart, 1992.

[4] Ahmed K. Elmagarmid (Hrsg.). *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, Inc., 1992.

[5] The Encina for AIX/6000 Product Family Library. *Distributed Transaction Service (TRAN).* IBM Corporation, 1991.

[6] Jim Gray. The Transaction Concept: Virtues and Limitations. In *Proceedings Conference on Very Large Databases; Cannes*, S. 144–154. IEEE Computer Society Press, September 1981.

[7] Roger Günthör. Evaluation of Dependency Rules. Technical report , IPVR Universität Stuttgart, 1991.

[8] IEEE. *IEEE P1003.4/D9*, 1992.

[9] Dieter Laube. Transaktionsorientierte Kommunikationsmechanismen für das verteilte Transaktionssystem APRICOTS. Diplomarbeit 865, IPVR Universität Stuttgart, 1992.

[10] X/Open Company Limited. *X/Open CAE Specification Distributed Transaction Processing: The XA Specification*. X/Open Company Ltd., U.K., 1991.

[11] Manuals. *Encina Transaction Processing System.* Transarc Corp., Pittsburgh, PA, 1991.

[12] Ioannis Masmanidis. Syntax und Compiler einer Sprache zur Beschreibung paralleler und verteilter Abläufe. Studienarbeit 897, IPVR Universität Stuttgart, 1990.

[13] Santosh Shrivastava, Stuart Wheater. Implementing Fault-Tolerating Distributed Applications Using Objects and Multi-Coloured Actions. In *Proceedings 10th International Conference on Distributed Computing Systems*, S. 203–210, 1990.

[14] U. Dayal, M. Hsu, R. Ladin. Organizing Long-Running Activities with Triggers and Transactions. *ACM SIGMOD Record*, S. 204–214, 1990.

[15] Helmut Wächter, Andreas Reuter. *The ConTract Model*. Chapter 7 in Database Transaction Models for Advanced Applications, Ahmed K. Elmagarmid (Ed.), 1992.

[16] Oliver Zellmer. Eine graphische Oberfläche für das verteilte Transaktionssystem APRICOTS. Studienarbeit 1111, IPVR Universität Stuttgart, 1992. 1992.

Studien and Diplomarbeiten are available via anonymous ftp:

host:	ftp.informatik.uni-stuttgart.de (129.69.211.2)
user:	anonymous
password:	anything (not empty)
path:	pub/apricots