

# The Impact of Concurrency Control on the Programming Model of ConTracts

Friedemann Schwenkreis, Andreas Reuter  
University of Stuttgart, IPVR  
Breitwiesenstr. 20-22  
D-70565 Stuttgart, Germany

Email: {reuter,schwenk}@informatik.uni-stuttgart.de

URL: <http://www.informatik.uni-stuttgart.de/ipvr/as/personen/{reuter,schwenk}.html>

**keywords:** transactions, concurrency control, workflow

## Abstract

*The ConTract model has been developed to extend the scope of classical (ACID) transactions to long-running executions such as workflows. A particular important aspect of the ConTract model is its approach to concurrency control. It uses semantic rather than syntactic synchronization, which influences both the programming style of long-running applications and the architecture of a ConTract processing system.*

*This paper will give an overview of the major aspects of the concurrency control approach of ConTracts. It will explain how applications can use the predicate based synchronization mechanism. Furthermore, the corresponding programming model will be introduced and a short introduction into the implementation is presented..*

## 1. Introduction

Long-running executions such as workflow processes require special concurrency control mechanisms. Because of their longevity, the classical serializability-based methods are not applicable [2, 3, 7]. For the ConTract model and its prototype implementation APRICOTS we introduced a semantic concurrency control mechanism which is based on invariance predicates [12, 10].

While implementing the approach several aspects have been elaborated. As it has turned out, the programming model is one of the most important issues besides performance, correctness and feasibility. Since the approach in ConTracts requires the explicit handling of concurrency control information, the programming model has to support the management of this information as far as possible. In contrast to the classical transaction model where concurrency control information is totally hidden from the application programmer, our approach forces the programmer to use this information. Furthermore, to relax isolation and overcome the restrictions of serializability, semantical rather than syntactical information is required.

The remainder of the paper is organized as follows: Section 2 briefly introduces the ConTract model and APRICOTS. In section 3, we will define the concurrency control mechanism used in ConTracts. Section 4 will present the corresponding programming model and section 5 will present the implementation. Section 6 will conclude with a summary and an outlook on future work.

## 2. ConTracts and APRICOTS

The ConTract model [12] was introduced to cope with the problems of long running applications. The basic idea was to organize such applications into several atomic units instead of modelling the whole process as a single classical ACID-transaction [2]. This resulted in the loss of the well-known properties of transactions for whole processes. Hence, the ConTract model introduced a set of new properties which are guaranteed for a ConTract (by a run time system):

- A ConTract, once started, will be finished in a finite amount of time (system guarantee for forward recovery).
- A ConTract guarantees its “compensability”. If requested by the application the execution can be seman-

tically reversed.

- A ConTract can protect data objects from concurrent access (of other ConTracts) outside of transactions.
- (Resource) Conflicts can be handled explicitly by the application.

To demonstrate the feasibility of the ConTract model, the concepts are implemented in a prototype system called APRICOTS [9]. The main emphasis in this implementation lies on the reliability of the system and the robustness of the execution of a ConTract. Therefore, the architecture was designed such that the system can tolerate any kind of a single component failure, both at the application and at the system level (see Figure 1).

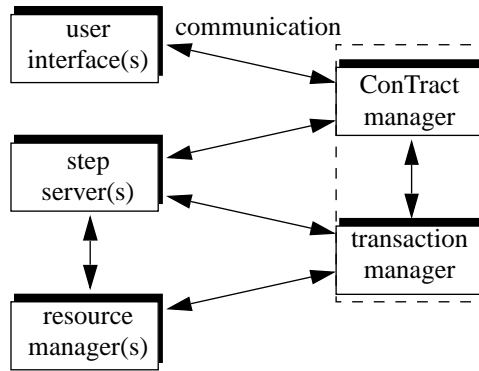


Figure 1: The basic architecture of APRICOTS

Since no adequate communication mechanism was available at the time when the prototype design started, a specialized communication system (CS) was developed. During the re-implementation of the system we decided to use a standard mechanism based on CORBA. Nowadays existing products (e.g. Orbix) support the features needed, such as transactional RPC's, migration of tasks, automatic start of servers and server classes in a transparent way.

## 2.1 An example

To motivate the features which will be presented in this paper, let us consider an example, which we will use throughout the paper. The process shown in Figure 2 is the simplified representation of a process of the marketing department of Mercedes Benz. It shows the way of distributing marketing brochures, flyers, etc. inside the company.

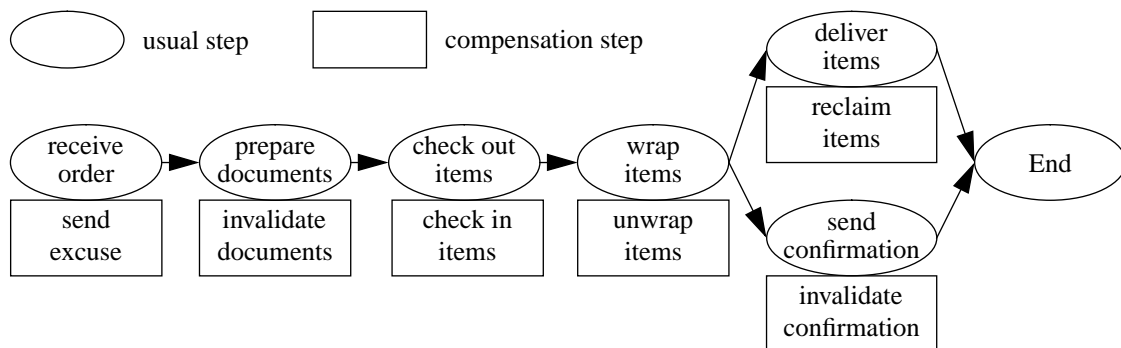


Figure 2: An example of a ConTract

1. First, the order information is collected, i.e. the delivery address, who is ordering what etc. It is also checked whether the material is available.

2. Shipping documents and related forms are prepared.
3. The ordered material is checked out from stock, and the corresponding database is updated.
4. The things are wrapped in the fourth step
5. The items are delivered, and a confirmation is sent to the marketing department; these two steps can execute in parallel.

To simplify the example, we have made the assumption that every step is modelled as a single transaction. Furthermore, parts of the script have been omitted (e.g. exception handling) for simplicity. Since each step runs as an atomic unit of work, the steps have to ensure that if they commit their task was successful. For instance, the step which is responsible for delivery must only commit if and only if it has received the acknowledgment from the recipient.

### 3. Concurrency control in APRICOTS

As mentioned before, conventional concurrency control mechanisms (CCM) are inadequate for long running executions. Furthermore, the classical correctness criterion of serializability is insufficient. In [10], we have introduced a predicate based concurrency control mechanism which is similar to [3]. It introduces a new correctness criterion for which we have shown its applicability in the area of long-lived executions. The major characteristics of this mechanism are:

- Application dependent degree of isolation by the definition of predicates.
- Support of compensation instead of “rollback”.
- Distinction between predicates to ensure correctness and predicates supporting the successful (forward) execution.
- Reliable (recoverable) concurrency control information outside of transactions.

#### 3.1 Ensuring correctness

The basic assumption of APRICOTS’ CCM is the special correctness criterion:

1. An execution is correct if a final state is reached, or
2. it is ensured that for every successfully finished activity, the compensating activity can be executed.

While executing a ConTract the first criterion is violated. Hence, the mechanism introduces predicates to protect objects needed for the compensation in order to ensure a state of these objects which allows the execution of the compensating actions.

In our example of Figure 2 this would result in the following constraints. Since not every compensation activity introduces a new aspect, only four cases are presented.

- After receiving the order it must be ensured that the collected data is available to be able to write a proper excuse letter (e.g. the address and the identification and the date of the order). Therefore, a predicate must ensure that the corresponding records are neither modified nor deleted (similar to an update lock).
- There are two ways to compensate the generation of the documents needed to process the order. It is either possible to delete the documents or it is wanted to keep them and just add a special tag which invalidates them. In both cases it is sufficient to protect the documents from deletion (similar to an existence lock).
- To ensure the compensation of the “check out” step a more sophisticated protection is needed. There can be two problems when trying to “check in” the items.  
The first problem is of the same class as in the previous case. To check in items the number of available

items is increased. Hence, the records of the items must be available.

The second problem is rather complicated. When items have to be put back in stock there must be room to store them. So, it must either be ensured that there is still some room at the original place where the items were at check out time. Alternatively, some spare room can be reserved to be able to store the items at least intermediately. This alternative brings in a new aspect of the concurrency control approach. It is possible that objects have to be protected which have not been accessed during the normal execution; they are only needed if the event of compensation.

- The compensation activity “unwrap items” does not have any requirements. So, there is no need to protect anything.

### 3.2 Further isolation needs

Besides the usage of predicates for guaranteeing correctness, predicates can also be used to ensure the “continuability” of an execution. The special property of these continuability predicates is that they do not have to be ensured until the end of a ConTract or the execution of a compensation activity. An application can decide at an arbitrary point in the execution to modify the predicate. Furthermore, the violation of these constraints do not violate the correctness of our execution which can be used for advanced policies for the management of these predicates (e.g. check-revalidate). This flexibility supports the negotiation about resources and cooperative work between ConTracts.

Our example provides this feature. The “receive order” step checks whether the ordered items are available or not, the “check out items” step does not need to check the availability again. On the other hand, there is no transactional protection of the corresponding records in the database. Hence, the predicates defined by the “receive order” step should ensure that the ordered amount of items is available. If the “check out items” step has finished successfully the predicates protecting the items in stock can be removed. Our example does not contain any exception handling steps. Anyway it would be easy to extend it to support a more flexible way to protect the items. For instance, a “conflict resolution step” can be introduced which is able to handle a violation of the constraints. If there are not enough available items when the “check out items” step has to be executed, the conflict resolution can send an order to the production department and wait until the items are available again.

### 3.3 Predicate types

The predicates used to define the needs of an application (workflow process) refer to *shared objects* accessible from several ConTracts running in parallel. For the further explanation of the predicates we introduce two operations on predicates: `establish(predicate)` and `check_predicate(predicate)`. The first operation checks whether the predicate holds and restricts future accesses. The second operation just checks the predicate.

Several types of predicates are distinguished to reflect the different ways to handle them:

- Simple predicates which are based on comparison operators (e.g. `<`, `>`, `=`).
- Composite predicates: predicates combined with logical operators (and, or)
- Simple predicates containing the functional combination of (shared, global) objects (e.g. `a+b`)
- Special predicates (e.g. “exists(object)” or “exclusive(object)”)

#### Simple predicates

Simple predicates are of the form:

`<context_object_descriptor> <comparison_operator> <context_variable | constant>`.

The `context_object_descriptor` references a shared object managed by a resource manager and the `context_variable` stands for a variable stored in the *context* of the ConTract. In section 4 we will introduce how `context_object_descriptors` are generated.

## Comparison operators

The set of comparison operators has been extended. New operators like “rel\_>” have been introduced because there are a lot of cases where the granularity of the access does not match the granularity of the accessed object. For example, the “receive order” steps checks whether the amount of available items is sufficient and wants to reserve some items regardless which. Usually, this is expressed by a predicate “No\_of\_items > x”. But the operator “>” is absolute, which can cause inconsistencies, i.e. if several ConTracts establish this predicate it is still sufficient that the value of “No\_of\_items” is x+1 and the needs of the ConTracts cannot be fulfilled. Therefore, the relative operators have been introduced. For instance, the semantics of the “*establish (o rel\_> k)*” operation is as follows:

- An object  $o$  for which the operator  $rel\_>$  is defined represents an interval, defined by the object’s constraints:  $o \in [lower\_bound(o), upper\_bound(o)]$ .
- If  $current\_value(o) \leq lower\_bound(o) + k$ : the operation returns false.
- If  $current\_value(o) > lower\_bound(o) + k$ : the operation increases the lower\_bound by  $k$  and returns true.

In contrast, the classical comparison operators “>” would set the value of the lower\_bound absolutely. It is obvious that further computation has to be done which is not described in this paper.

The introduced relative comparison operators are very similar to other interval based methods [6, 5]. However, the mechanism is not limited to numerical data types. It can be used for any object which supports the introduced interfaces. As an example, one can think about the reservation of space for the “check in items” step of our example. The corresponding predicate can be of the following form:

$storage\_room \ rel\_> \ (10 \ yards, \ 12 \ yards),$

i.e. reserve an area which is greater than 10 yards in width and 12 yards in length.

## Virtual objects

Another degree of complexity comes in if the functional combination of (global) objects is used in the left part of the introduced simple predicates. In our example this could happen if a department from Switzerland orders some brochures. Since Switzerland is a multilingual country they do not care too much about the language of the brochures as long as the language is either german or french. This would result in a predicate like “*No\_of\_german\_brochures + No\_of\_french\_brochures rel\_> x*”. In case of arithmetic functions and numerical data types the evaluation is relatively simple. If arbitrary functions and data types shall be supported it gets rather complicated. Furthermore, it is a bad approach to keep functional combinations inside of predicates. Since, the same functional combination of objects can occur multiple times and has to be used potentially at multiple locations, it is reasonable to separate the evaluation of functional combinations from the evaluation of predicates. Hence, the concept of “*virtual objects*” has been introduced [10]. The corresponding mechanism extracts the functional combination of objects from the predicate and replaces it with a virtual object. In our example this would result in a modified predicate of the form: “*virtual\_object rel\_> x*”. The *virtual\_object* implements the functional combination as well as the communication with the real objects. So, several optimizing policies for the evaluation of the functional combination can be applied without a change of the predicate management.

## Special keywords

As motivated by our example it is sometimes (as we think very often) just needed to establish a special lock, like the existence lock, on an object. Therefore, the special keywords have been introduced, which can be implemented directly as recoverable locks with the well-known mechanisms from the database area. We propose a basic set of special keywords which should be supported by every object: *exists*, *shared*, *update*, *exclu-*

sive, insert. It is obvious, that these types of locks are already supported by today's database systems but there is still to do some work to make them recoverable.

More complex objects can introduce advanced locks (advanced keywords) like “*update(object) share\_with(group)*”, which means that everybody is allowed to read the object, but only members of the group can update the object.

#### 4. The programming model

A major difference between ConContracts and other “extended transaction models” [1] is the fact that the model of ConContracts introduces a programming model. The basic assumption of this programming model is that there are at least two levels of programming: the “script-level” and the “step-level” (the programming of resource managers is not taken into account). At the step level usual programming techniques are used (e.g. C++-programming) to program the basic activities and to provide the transactional behavior of steps. Programming on the script level means to define a control flow between steps, to define transactional borders and so on.

The programming model assumes that the two levels are almost independent from each other in the sense that the programmer of a script only uses predefined (pre-programmed) steps and the definition of the script has no impact on the definition of the steps (but not vice versa). Hence, a script programmer needs knowledge about the semantics of steps and their interfaces (parameters) which is supported by a graphical programming environment [11].

##### 4.1 Using knowledge of the step level

Since the programmer of a script has only the knowledge about objects of the *context* [8] (the specialized data model of ConContracts), he or she cannot define invariants without being provided with additional knowledge about (global) shared objects accessed by the steps (and their compensating steps). Hence, steps have to “export” information about the objects on which invariants can be defined:

1. The programmer of a step generates a set of so-called *object descriptors*, which can be used at the script level.
2. Step programmers can add a set of so-called *invariant templates* to the definition of a step which can be used to derive invariants.
3. The script programmer can use object descriptors like variables of the context. This will be supported by the programming environment. In addition, functional combinations of objects will be identified and replaced by corresponding *virtual object descriptors*.

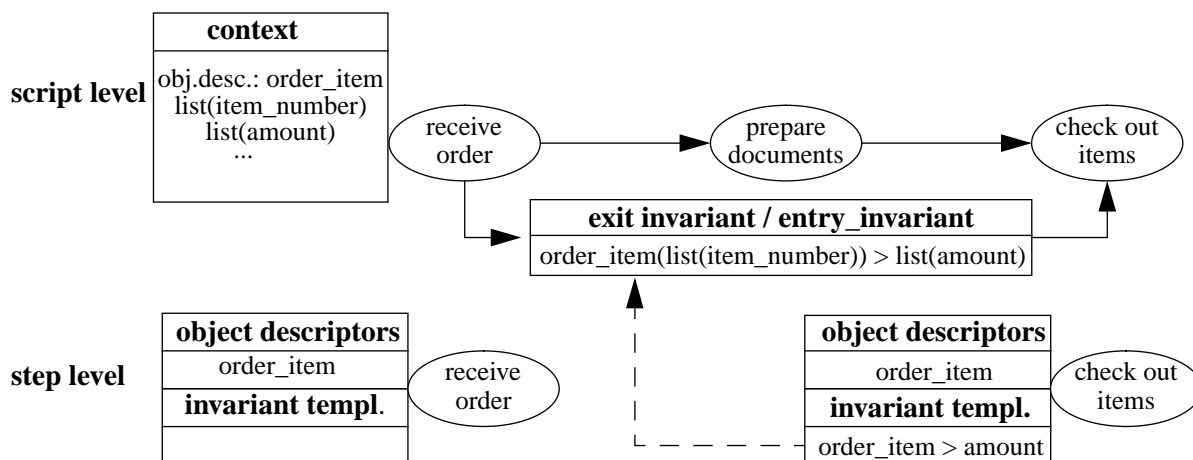


Figure 3: An example for programming invariants

The invariant templates express the needs of a step to run successfully in a more or less abstract way. A script programmer can use these templates by adapting them (in the definition of the script) to his or her specific case (see figure 3).

Object descriptors contain information about how to access the associated object and the type of the object. Since the type of the object defines the allowed operations and comparison operators, the programming environment can check whether an invariant is well-formed or not (taking the template into account). In addition, object descriptors can have an associated *identifier list* as well as a *value list* which allows a more generic usage. This means, that an object identifier can represent a whole set of object instances. To define or refer to a set of specific instances at run-time a list of identifier values and a list of object values has to be generated and stored in the context.

In our example, the “receive order” step would export an object identifier “order\_item” with an associated identifier list of the type “item\_number”. A possible invariant would be “order\_item(list(item\_number)) > list(amounts)” which has to be translated by the run-time environment into the proper predicates. For instance, if two brochures with item number 101 and four flyers with item number 504 have been ordered, the list of item numbers would be (101, 504) and the list of amounts would be (2,4). The run-time environment will then translate the predicate *order\_item(101,504) rel\_> (2,4)* into a composite predicate:

$$order\_item(101) rel\_> 2 \text{ AND } order\_item(504) rel\_> 4$$

## 4.2 Addressing objects

A prerequisite for using object descriptors in the way described in the previous section is the uniqueness of each descriptor. In the above example, the programming environment must be able to detect that the object descriptor “order\_item” of step “receive order” represents the same object as the object descriptor “order\_item” of the step “check out items”. Since the whole system has been moved to CORBA, a mixture of DCE oriented and object-oriented naming schemes is used to refer to global objects:

$$RM\_name/Internalpath/object\_name.[object\_identifier\_name.identifiervalue]$$

The resource manager name (RM\_name) and the internal path (Internalpath) can also be composite strings separated by a dot. In our example, where the information about available items is stored in a (relational) database, the corresponding name defined by a object\_descriptor may look like:

$$Company/stock\_database.item\_table/no\_of\_items$$

To resolve object names when identifier lists are used, the run-time system extends the name by the identifier name and the value. The first predicate of the composite predicate of the previous section would then be:

$$Company/stock\_database.item\_table/no\_of\_items.item\_number.101 rel\_> 2$$

This approach seems to be very simple, but that is only true for simple datatypes. In case of more complex datatypes a generic mechanism has to be used which translates the value to a proper string representation. Fortunately, CORBA has built-in mechanisms which implement such a feature.

## 4.3 Dynamic aspects of invariants

### Context related problems

Since the data model of ConTracts (the *context*) is defined as an append only stable storage, several versions of a context variable exist at run-time [8]. Now, if a context variable is used to define an (exit) invariant, the question arises which version of the variable has to be used when the invariant is checked again (entry invariant). To be able to identify the version of the context variable which has to be used for the evaluation of the predicate, a proper addressing schema is needed. Hence, a programmer must have the possibility to use the same addressing schema as introduced for the programming of the data flow, while programming invariants. He or she can define certain attributes which identify the point in the execution where the wanted version of

the context element was available:

1. the name of the context variable
2. the name of the step instance after which the wanted version was available
3. a value of the loop counter when the step was executed inside of a loop (loop index)
4. an identifier for the thread when the same step is used in multiple (concurrent) threads (parallel index).

The name of a step's instance is a unique name inside of a ConTract, i.e. if the same step (at the step level) is used multiple times at the script level, the programming system generates a unique names for each instance. Generic constants can be used for the cases 2, 3 and four . That means the systems default is to use "CURRENT" if the programmer does not define anything. Hence, always the most recent version of a context variable is accessed. Even this feature is very nice for programming the data flow, it is unwanted while programming invariants. To be more specific, if a step establishes an invariant containing values of context variables, it must be ensured that the entry invariant of another step, which refers to the prior established exit invariant, still contains the same values even if new versions of the context variable have been generated.

To support this need at a programming level, the programming system offers the possibility to define invariants by referencing prior defined invariants. In the introduced example the programmer does not need to explicitly define the (entry) invariant of the step "check out items". Instead, he or she just inserts a reference to the prior defined (exit) invariant of the step "receive order". Consequently, invariants themselves become (special) objects of the context.

### **Modification of invariants**

Invariants must be maintained only until the step is executed which required the protected state of objects or until it is guaranteed that the corresponding step will never be executed. That means, that invariants which ensure a certain object state for a compensation step can be removed if the compensation step has been executed successfully or the whole ConTract has been finished successfully. But this is only the simple case. Since a programmer can also define invariants to support the successful execution (see section 3.2), the system has to remove and establish invariants during the normal execution.

The invariant approach is based on the assumption that if a step has certain requirements regarding the state of global objects, other steps have to ensure these states before the execution of the step which requires these states (otherwise the invariant is just a "ping" if the step can run or not). In result, we have a generalized form of the producer/consumer principle: steps which define an exit-invariant are producers - they establish invariants; steps which define an entry-invariant are consumers and just check invariants. However, a step which refers to an entry-invariant is not necessarily the only step which references that invariant. Hence, the following approach is made to release invariants as early as possible, but reducing the programming overhead as much as possible:

- The programming system allows to reference exit-invariants while defining entry-invariants
- The programming system generates delete calls for invariants if every step which has referenced the invariant has finished its work successfully. That means that the script contains explicit calls to delete an invariant.
- The run-time system deletes the invariants if it is guaranteed that steps which still reference the invariant will never be executed. This is the case if the ConTract has finished its execution, or when a path of the execution is detected which can never be followed (dead path elimination).

## 5. The current implementation

The past six months have been used to re-implement APRICOTS. The whole system is now based on CORBA, CORBA services (COS naming service, O TS, etc.) and the Encina (T ransarc) log service. The servers are implemented using C++ while the clients (user interfaces) are based on Java. DB2 is used as an example for a resource manager but almost any relational database can be used.

### 5.1 Programming support

The current implementation supports programmers only on the script level. That means that the programming (definition) tool for scripts support the import of invariant templates and the mapping of generic elements of object descriptors on context variables.

The support on the level of steps is not v ery far developed. There is just a v ery basic tool which allows the definition of invariant templates such that they can be used on the script level.

### 5.2 Run-time support

Due to fault-tolerance reasons, we decided to locate the invariant management at the step servers. Hence, the ConTract manager does not need to know anything about the details of invariants. Its only task is to pass the invariants to the appropriate step servers. There is just one case where a ConTract manager may manipulate an invariant. It has the right to change the policy of the invariant, i.e. if an invariant is a “want” invariant and there was a resource conflict. The contract manager may decide to change the policy of the invariant from “ensured” to “check/revalidate”. Thus, a step server which references that invariant (it’s the entry invariant of that step server) is forced to check again if the invariant holds.

#### Step servers

Step servers must provide a specific set of interface functions:

- **stepService** request methods:  
invoked by the ConTract manager to pass a request (and invariants). These methods have a standard and a step server specific part of their interface. Since the interface definition is available at run-time (script definition) and not necessarily at compile time of the ConTract manager, the ConTract manager uses the dynamic invocation interface of CORBA to invoke such a method.
- **dropInvariant**:  
a standard method for the dynamic invariant handling (invoked by the ConTract manager).
- **endOfConTract**:  
a standard method which informs a step server to drop all invariants (invoked by the ConTract manager).
- **checkInvariant**:  
invoked by other step servers to check whether their entry invariant still holds (if the policy is check/revalidate).
- **other transaction specific interfaces**:  
since a step server is at least a (CTS-)transactional object it must provide an interface for the transaction service.

Now, if a step server receives a step service request it also receives its entry and exit invariants. Since an entry invariant must be established by a step executed before, the invariant is just a reference of an invariant which was established by another step. If the policy is “check/revalidate” the step invokes the **checkInvariant** method of the previous step to ensure that the invariant still holds. If the **checkInvariant** returns false, the step returns a special error code to the ConTract manager. If the policy was not check/revalidate the step can proceed without checking the invariant again.

The exit invariant of a step is processed at the end of the execution of a step. The step server splits the invariant into peaces which belong to a single resource manager . Next, the step server creates the necessary virtual objects and establishes the invariants, i.e. the predicates are send to the resource managers which check the predicates and ensure their validity.

## 6. Summary and prospect

In this paper we have presented the impact of concurrency control on the programming model of APRICOTS/ConTracts. Since long running applications do need specialized features at the programming level as well as for concurrency control, the concurrency control mechanisms have to be supported by the programming model. Even the presented approach is focused on a ConT ract system, we strongly believe that concurrency control for long running applications will always influence the programming level. Hence, the corresponding models should take care of this point.

There are still a lot of questions re garding the integration of a ConT ract processing systems with e xisting resource managers like database systems. Since today's transactional resource managers do not support recoverable locks, they cannot be integrated directly. Instead, a wrapper has to be written, which establishes the locks again at restart. During that phase the local access of resource managers (not issued by a ConT ract) must be prevented.

In our future work we will proceed with the definition of a proper architecture and inte grate the concurrency control mechanism with APRICOTS. Then, we will test the system with real workflow applications to measure performance and the applicability of our approach.

## References

- [1] Ahmed K. Elmagarmid (ed.): Database Transaction Models for Advanced Applications, Morgan Kaufmann Publishers, 1992.
- [2] Jim Gray: The Transaction Concept: Virtues and Limitations, Proceedings of the 7th International Conference on Very Large Data Bases, Cannes (F), 1981.
- [3] Henry F. Korth, Greg Speegle: Formal Aspects of Concurrency Control in Long-Duration Transaction Systems Using the NT/PV Model, ACM Transactions on Database Systems, Vol. 19 No.3, 1994.
- [4] Vijay Kumar (ed.): Performance of Concurrency Control in Centralized Database Systems, Prentice Hall Inc., 1996.
- [5] Patrick E. O'Neil: The Escrow Transactional Method, ACM Transactions on Database Systems, Vol. 11 No.4,S. 405-430, 1986.
- [6] Andreas Reuter: Concurrency on High-Traffic Data Elements, Proceeding of the ACM Symposium on Principles of Database Systems, 1982
- [7] Andreas Reuter: An Analytic Model of Transaction Interference: in [4] Chap. 4.
- [8] Andreas Reuter, Friedemann Schwenkreis: ConTracts - A Low-Level Mechanism for Building General-Purpose Workflow Management Systems, IEEE Bulletin of the Technical Committee on Data Engineering, Vol. 18 No. 1, 1995.
- [9] Friedemann Schwenkreis: APRICOTS - A Prototype Implementation of a ConContract System: Management of the Control Flow and the Communication System, Proceedings of the 12th IEEE Symposium on Reliable Distributed Systems, Princeton (NJ, USA), 1993.
- [10] Friedemann Schwenkreis, Andreas Reuter: Synchronizing Long-Lived Computations, in [4] Chap. 12.
- [11] Friedemann Schwenkreis: APRICOTS - a workflow programming environment, in Proceedings of the 6th International Workshop on High Performance Transaction Systems (HPTS), Pacific Grove (Ca, USA), 1995.
- [12] Helmut Wächter, Andreas Reuter: The ConContract Model, in [1] Chap. 7.