To appear in:
N.W. Paton, M.W. Williams (eds.): Rules in Database Systems. Workshops in Computing, Springer Verlag.

Rule-Based Implementation of Transaction Model Specifications

Andreas Geppert
Klaus R. Dittrich
Institut für Informatik, Universität Zürich
Zürich, Switzerland

Abstract

We propose an active-mechanism based approach for the construction of database management systems (DBMS). We conceive a DBMS to be composed out of brokers (which realize subsystems), where a broker is defined by a set of production rules. In this setting, we show how specifications of transaction models can be realized. To that end, we apply a rule-based approach for the transformation of (logical) specification rules into production rules, which in turn describe the behaviour of the transaction management broker.

1 Introduction and Motivation

During the last decade, a significant number of database management systems (DBMSs) and persistent object systems have been realized for rather different application domains. Due to the diverging requirements of these application domains, no consensus exists —and most likely will never exist— for one, or even a small number, of DBMSs that are adequate for all these application domains. Based on this observation, we expect a variety of DBMSs to coexist in the future.

As soon as more and more DBMSs have to be realized, the development of a *construction methodology* for DBMS becomes an urgent research issue. This is also the case with the development of toolkits and support tools for DBMS construction. The two most important characteristics of a successful construction methodology are:

- it covers all the relevant aspects of a DBMS, and
- it supports all the phases of DBMS construction.

While the major part of research in the field of extensible database systems [1] investigated support for the realization of data models, other aspects like transaction management, integrity enforcement, and so forth which are equally important are not addressed. Furthermore, extensible database systems concentrated on supporting *implementation*, while analysis and design are completely left to the user (i.e., the database system implementor, DBI).

Significant work has been done in the areas of specification and implementation of transaction management. For example, ACTA[1] [2] is useful for the specification of (extended) transaction models. Furthermore, toolkits are available for realizing transaction managers [3] (i.e., class libraries containing primitives for the implementation of transaction management). However, the work on ACTA does not address implementation so far, and toolkits alone do not support analysis and design very well. Hence, the (crucial) mapping from transaction model specifications to realizations is still an open problem.

In this paper, we address the problem of realizing ACTA specifications in the context of a DBMS construction method. In the KIDS[2] project [4], we investigate such a method based on large-scale reuse of software artifacts. In order to make reusability successful, the construction method makes (re)use of not only code, but also analysis and design information [5]. Roughly speaking, various *aspects* (e.g., data model, transaction model, integrity model) of a full-fledged DBMS are specified using aspect-specific languages. Based on these specifications, *subsystems* are realized by means of configuration/composition of reusable primitives, or by generation. These subsystems are plugged into an architecture skeleton (thereby making the DBMS architecture itself also subject to reuse).

As for transaction management, we conceive (sets of) ACTA-rules as constraints on correct histories, i.e., correct interleavings of operations issued by concurrently executing transactions. Operations are usually transaction events (like commit , abort) or database operations (e.g., insert, update, and so on). In order to ensure correctness, we translate ACTA-rules into (a simple form of) production rules (e.g., [6], [7]). In contrast to the majority of other work, however, production rules are applied as a construction and realization paradigm for *DBMS-internal* purposes. In other words, production rules are defined during the construction of a subsystem (of a DBMS) and are not necessarily visible to schema designers or end-users.

The remainder of this paper is organized as follows: in the following section, a short overview (as far as necessary for the understanding of this paper) of our understanding of a subsystem is given. Section 3 contains a short description of the ACTA framework. In section 4, we describe the transformation of ACTA specifications into production rules.

2 The Broker/Services Model

Our ultimate goal is to support the construction of database management systems (DBMSs) in such a manner that a DBMS implementor (DBI) is assisted throughout the analysis, design, and realization of (aspects of) a DBMS. Currently, we address three aspects: data models, transaction management, and integrity enforcement. Each of these aspects is realized by a *subsystem*. The *analysis* of one aspect results in a formal (requirements) specification, written in an aspect-specific language. *Design* refers to the development of the structure of the subsystem and its relationships to other subsystems, while during *realization*, classes and objects are determined that implement the subsystem's behaviour. In principle, realization can be done through the configuration or generation of components.

1. "Actions" in Latin
2. ***K***ernel-machine based ***I***mplementation of ***D***atabase Management ***S***ystems

For the construction of the various subsystems of a DBMS, we have the following requirements from an architectural point of view:

- While the independent and parallel development of various subsystems has to be supported, these are typically not completely independent from each other. Hence, we need a flexible model for interfacing subsystems. Especially, if (parts of) subsystems are generated, they will have to make assumptions on other subsystems, without knowledge of their concrete realization or of the principles guiding other generators.
- The stepwise realization of subsystems and control flow between subsystems has to be supported.
- A uniform model for the execution of DBMS services and the description of their semantics is necessary.

In order to satisfy these requirements, each subsystem is governed by a so-called *broker*, which is the only visible part of a subsystem. A broker is responsible for specific *services* and "knows" which actions to perform for requested services. At execution time, services are requested in that *events* are raised. These events are detected by the responsible broker, which then decides whether actions have to be performed and eventually executes them. One part of a reaction can be a *reply*, which is then caught by the object that raised the event. Other kind of reactions comprise propagation of events to subcomponents or the generation of additional events. At construction time, the designer specifies events which will be raised at execution time. Hence, in order to implement the control flow (between subsystems), we have to determine responsible brokers and to define how the respective broker reacts to the event. This process continues recursively if the (reacting) broker in turn raises events.

Brokers are defined by a set of ECA-rules (or equivalently, production rules) [6, 7], where each rule defines an event which the broker is responsible for, the corresponding reaction, and a condition that determines whether the reaction is performed or not. For our purposes, *simple events* (actually, *abstract events* [7]) are sufficient, whereas composite events are not necessary. However, events can have parameters which have to be supplied when the event is raised. Production rules have the following syntax:

```
ON    event (arguments)
IF    condition
DO    action
```

where the condition can be omitted if it is assumed to be always true:

```
ON    event(arguments)
DO    action .
```

In this way, a broker offers a set of standard services; other components interacting with a broker only need to know the services, but not concrete interfaces or realization details. A broker, however, has to know (or to determine, during construction) how to realize a specific service. Hence, similar to the common object request broker architecture (CORBA [8]), a broker can be seen as a mediator between clients and servers.

In this paper, we focus on the realization of transaction management brokers, i.e., the mapping of a specification language for transaction models to (production) rules implemented by the transaction broker.

3 A Short Overview of ACTA

The intention of this section is to introduce ACTA, since a rough understanding of this specification framework is necessary for the comprehension of the translation approach explained below.

ACTA [2, 9] is a framework for the specification of extended transaction models based on first-order logic. Within this framework, *transaction types* can be specified, whereby a transaction type intensionally describes a set of transaction instances that share structure and behaviour. A transaction instance issues *events*, where the set of all event types can be further classified into *transaction events* (i.e., begin , commit , abort , split , join [10] and so forth), *object events* (i.e., data manipulation actions), and the like. The sequence of all occurred event instances forms a *history*, whereby the history up to a given point in time *t* is referred to as the *current history* H_{ct}. Two essential predicates are $e \in H$, where e is an event and H is a history (e is an element of history H), and $e_1 \rightarrow e_2$ (event e_1 precedes event e_2).

Typically, (parts of) the semantics of one transaction type depends on its relationships to other types. These relationships are captured by ACTA through the notion of *dependency*. Usually, a dependency is an implication that constrains the occurrence or order of events of two transactions (see Table 1). As an example, take an abort dependency between transactions t_1 and t_2: if t_1 aborts, t_2 has to abort as well, i.e., $\mathrm{abort}(t_1) \in H_{ct} \Longrightarrow \mathrm{abort}(t_2) \in H_{ct}$. Dependencies can arise due to structure (e.g., a subtransaction is weak-abort dependent on its parent transaction) or due to behaviour. In the latter case, dependencies can also be formed due to *object events* of the related transactions. For instance, if one transaction executes an operation on an (uncommitted) object, it is usually abort-dependent on the transaction that modified the object most recently.

Transactions perform operations (as defined by the data manipulation language) on objects. ACTA abstracts from both, the concrete set of operations and the notion of "object". However, we are interested in rules which constrain the access of transac tions to objects and which define dependencies due to operations performed on objects on behalf of transactions. These rules typically use the definition of the following sets and functions:

- The *view set* of a transaction specifies which objects are visible (i.e., can be accessed and manipulated) to it. The view of a transaction can be defined in terms of the current history, other transactions, and existing dependencies.
- The *conflict set* of a transaction defines the operations that might conflict with the operations issued by the transaction. Like the view set, the conflict set of a transaction can be defined in terms of the current history, other transactions, and existing dependencies. If two operations p and q conflict (i.e., the respective return values depend on their execution order), the predicate conflict(p,q) is *true*. The definition of this predicate is assumed to be predefined for a given transaction

Name	Definition	Meaning
Strong-Commit: t_2 SCD t_1	$commit(t_1) \in H \Rightarrow commit(t_2) \in H$	if t_1 commits, t_2 commits as well
Abort: t_2 AD t_1	$abort(t_1) \in H \Rightarrow abort(t_2) \in H$	if t_1 aborts, t_2 aborts as well
Weak-Abort: t_1 WD t_2	$abort(t_2) \in H \Rightarrow (\neg(commit(t_1) \rightarrow abort(t_2)) \Rightarrow abort(t_1) \in H)$	if t_2 aborts and t_1 has not yet committed, t_1 aborts as well
Force-Commit-on-Abort: t_1 CMD t_2	$abort(t_2) \in H \Rightarrow commit(t_1) \in H$	if t_2 aborts, t_1 has to commit
Begin-on-Commit: t_2 BCD t_1	$begin(t_2) \in H \Rightarrow (commit(t_1) \rightarrow begin(t_2))$	t_2 can begin only after t_1 has committed
Begin-on-Abort: t_2 BAD t_1	$begin(t_2) \in H \Rightarrow (abort(t_1) \rightarrow begin(t_2))$	t_2 can begin only after t_1 has aborted
Weak-Begin-on-Commit: t_2 WCD t_1	$begin(t_2) \in H \Rightarrow (commit(t_1) \in H \Rightarrow (commit(t_1) \rightarrow begin(t_2))$	if t_2 has already started and t_1 has yet committed, then t_2 started after t_1 has committed

Table 1: Sample ACTA-Dependencies

model and an underlying data model. If two operations conflict and p_{t1} has been executed before q_{t2}, the predicate $C(t_1, t_2)$ evaluates to *true* (i.e., "t_1 is serialized before t_2"). C^+ is the transitive closure of C.

- The *access set* of a transaction identifies those operations for whose commit or abort the transaction is responsible. These responsibilities can be re-assigned to other transactions by means of *delegation*.

Given these constructs, a correctness criterion can be specified in that restrictions on transactions and objects are defined and in that dependencies are established. As an example for the first, a condition for a transaction to commit is $\neg C^+(t, t)$ (i.e., the absence of cycles in the serialization graph [11]), while an example for the second is an abort dependency (formed between t_1 and t_2) established when p_{t1} is executed before q_{t2} and both conflict.

4 Mapping ACTA Specifications Into Production Rules

In this section, we show how the rules of an ACTA specification can be mapped to production rules to be realized by a transaction broker.

4.1 Problems and General Solution

For realizing ACTA specifications, we are faced with two general problems:

1. ACTA-rules are based on logic (*deductive rules*), while the execution model is based on *production rules*.
2. Histories grow dynamically, while it is not feasible to wait until a history is complete and to decide then whether the complete history is correct.

As for the former, ACTA specifications restrict the set of correct histories (by means of rules). The problem here is that ACTA specifications do not tell us how to *ensure* correctness. They do neither specify *when* to check rules nor *how to react* in case of inconsistencies. The second, yet related problem is the sometimes underlying assumption that histories are complete, i.e., contain all the events issued by the transactions (including one termination event for each transaction). Again, naturally, it is not practical to reason only about complete histories, since in case of incorrect histories the entire history might need to be backed out. In fact, we have to detect (and avoid) inconsistent histories as soon as they occur, whereby the problem is that we have no knowledge about future events.

Similar to approaches for integrity enforcement, our solution translates deductive rules into production rules. This translation is again rule-based and is specified by *transformation rules*. Resulting production rules will ensure that the current history can be extended to a correct, complete history (in terms of transaction theory: the current history is a *prefix* of a correct history).

The general approach for transformation rules is to first rewrite ACTA-rules so that they obey a canonical form; namely, implications with a single ACTA event as antecedent (left hand side of the implication). In this way, we derive events of resulting production rules: namely, an event that lets the antecedent of the (ACTA) rule evaluate to *true*. In this case, we know that the consequent of the (ACTA) implication also has to be *true*. Hence, a reaction has to be performed whenever the consequent is *false*. In order to avoid this incorrect case, two reactions are possible: we can either avoid the antecedent to get *true* or enforce the consequent to become *true* as well. To that end, two strategies for reactions of production rules are possible:

- the preservation of the validity of (ACTA) rules can be *enforced* by raising or propagating events,
- validity of rules can be preserved by *rejecting* (i.e., ignoring) events.

Each of the alternatives already determines the kind of action to be performed upon the occurrence of events. In the first case, we ensure that a given ACTA rule evaluates to *true* since —by means of raising additional events— we guarantee that consequents of rules evaluate to *true*. As an example, in this case of abort dependencies, the consequent specifies that other (dependent) transactions also have to abort.

Hence, since they cannot come to a successful end, it is appropriate to abort them at the current point in time.

Nevertheless, it might not be possible to guarantee the preservation of the validity of some ACTA-rules (dependencies). For instance, assume the presence of a strong-commit dependency for compensating transactions. If the compensated transaction committed and the parent transaction aborts, the compensating transaction has to (execute and to) commit as well. However, it would not make sense to enforce the commitment of the compensating transaction (at the time the compensated transation commits), nor would it be adequate to disallow the compensated transaction to commit (since we cannot know whether the compensating transaction will ever execute). Hence, in this case we will generate production rules that fire on the *negation* of the required event and react adequately. In the example of commit dependencies, two rules can be generated: one that fires if an internal abort (e.g., due to deadlock) occurs and then restarts the compensating transaction as reaction, and one production rule that is triggered by an "external" abort (e.g., user intervention, semantic incorrectness in the sense of integrity violation) and reacts by starting a contingency transaction (alternative transaction) [12].

In the second case, we cannot preserve the validity of the rule by performing actions that would let the consequent evaluate to *true*. In rejecting the event, both parts of the implication evaluate to *false* and the rule is valid again. Hereby, *rejecting* an event means to act as if the event had never occurred. Consequently, the event is not an element of the current history. The event can be raised once more later on and can then be allowed if the corresponding rule is preserved. For example, assume a rule that specifies that nested transactions cannot commit until all their subtransactions have committed or aborted. If a nested transaction tries to commit while at least one of its subtransactions is still active, the commit request will be rejected (and we can imagine a corresponding notification to the application). Nevertheless, the nested transaction can try again (i.e., re-raise the event) at a later point in time.

Consequently, event occurrences are classified into two classes. Depending on the current history and the set of rules an event occurrence can be either *legal* or *illegal*. In the latter case, there is at least one rule that will reject that event. This distinction also implies a specific execution (or event handling) strategy: no rule is allowed to trigger upon the occurrence of an event until it is determined whether the event is legal at all. Additionally, as a prerequisite for realizing ACTA specifications, we classify ACTA dependencies as being either *enforcing* or *rejecting*. [3]

4.2 Assumptions

Without loss of generality, we make the following assumptions.

First, we assume that a *state* is attached to each transaction. Typically, any transaction is in one of the states *initialized*, *active*, *committed*, or *aborted*[4]. If we want to express that a transaction is no longer active, irrespective of whether it aborted or

---

3. This is in contrast to [13], where *events* are classified in rejectable, enforcible, and delayable ones.

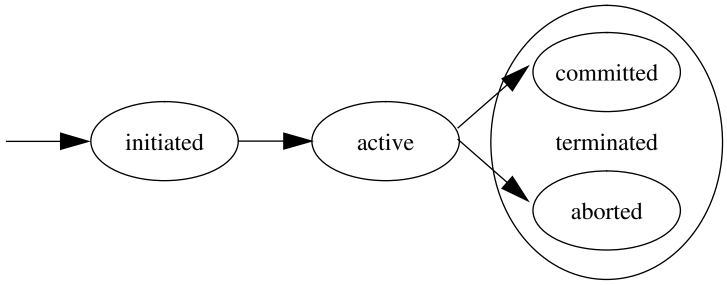

Figure 1 Transaction States

committed, we refer to the state as *terminated* (Figure 1*)*. Likewise, *in-progress(t)* means that transaction *t* has not yet terminated. If a transaction *t* is in state *s*, the predicate *s(t)* evaluates to *true*. The state predicates later on will help to avoid the explicit representation of histories.

Further predicates and functions represent the structure of transactions. For instance, we have to formally represent parent/child-relationships of nested transactions, as well as the relationship between component transactions and compensating transactions in the Saga model [15]. Since the definitions of these functions are rather obvious, we omit them here for the sake of brevity. Also, we assume behaviour such as delegation, transaction rollback, commit of changes, and the like are implemented by various (transaction) classes.

Given these assumptions, the mapping of ACTA-rules into production rules is organized as follows:

1. The mapping of dependencies to production rules, and
2. The mapping of constraints on operations on objects to production rules.

The remainder of this section will be organized along these lines.

4.3 Realizing Transaction Dependencies Through Production Rules

Given the above distinction of rules, inter-transaction dependencies are generally realized in three steps:

- possibly rewrite ACTA-rules such that transformation rules can be applied,
- apply transformation rules which generate production rules for establishing dependencies,
- apply transformation rules generating production rules that fire when dependencies have to be preserved (for brevity, we say that “a dependency is triggered” in this case).

4. Additional states may be necessary, e.g., for distributed transactions or for realization of coupling modes in an active DBMS [14]. However, the aforementioned states are sufficient for now.

As examples for the latter two, assume an abort dependency as given in Table 1. Establishing the dependency for two specific transactions means simply recording the dependency. None of the aborts is actually performed at this point in time. However, the dependency is triggered as soon as the first transaction aborts and the abort has to be propagated to the second transaction.

4.3.1 Preparing ACTA-Rules for Transformation

Generally, we assume that both dependency-establishing rules as well as dependency-preserving rules are given as logical implications, and that the antecedent is a single event occurrence. In order to obtain this structure, ACTA-rules may need to be rewritten using well-known logical rules. For instance, assume an ACTA rule of the form ($e_1 \wedge e_2 \Rightarrow c$), where c is a condition. This rule will be rewritten into two ACTA-rules as follows: $e_1 \Rightarrow (e_2 \Rightarrow c)$ and $e_2 \Rightarrow (e_1 \Rightarrow c)$. Later on, e_1 and e_2 will play the role of triggering events in the two resulting production rules, respectively. Additionally, ACTA events can still occur in consequents of rules, e.g., transaction events. These events will be replaced by predicates, e.g., on the state of the respective transaction. For instance, if the consequent contains $abort(t) \in H$, we will replace it by the state predicate $aborted(t)$. By means of this replacement, we know that the specified event has occurred (and hence is an element of the history). Therefore, we do not have to represent (and check) the history explicitly.

4.3.2 Production Rules for Establishing Dependencies

In general, a transformation rule is a pair *(i, r)*, where *i* is an implication (ACTA rule) and *r* is a production rule.

The general scheme for the transformation of dependency-establishing rules is as follows:

ACTA:	$post(e_1(t_1)) \Rightarrow ((t_1\ DEP\ t_2) \in DepSet_{ct})$
Production Rule:	ON $e_1(t_1)$ DO raise $(dep(DEP, t_1, t_2))$

Rule 1 Production Rule for Establishing Dependencies

The meaning of the ACTA rule is that after event e_1 has occurred, the dependency (t_1 DEP t_2) has to be in the set of current dependencies. The semantics of the production rule generated by (the transformation) Rule 1 are to establish the dependency as soon as the antecedent of the defining ACTA rule evaluates to *true*. At execution time, an event is raised and caught by the transaction broker, i.e., the predefined reaction of the event dep is to register the dependency (DEP). The notion of the current set of dependencies ($DepSet_{ct}$) is implicit and therefore does not have to occur in the production rule definition. Note that the ACTA rule is rather simple in this case, since its consequent is only a predicate. If the consequent in turn consists of an implication, we apply Rule 2 where c is a condition that might contain predicates over t_1 and t_2

ACTA: $post(e_1(t_1)) \Rightarrow (c \Rightarrow (t_1 \text{ DEP } t_2) \in DepSet_{ct})$

Production Rule:
ON $e_1(t_1)$
IF c
DO raise (dep(DEP, t_1, t_2))

Rule 2 Establishing Dependencies with Conditions

4.3.3 Production Rules for Triggering Dependencies

Once dependencies are registered, they trigger as soon as the corresponding event occurs. For each case we define separate transformation rules. The first case, where the preservation of the validity of the consequent can be enforced, is termed *enforcement*. The second case, where the consequent cannot be enforced and the triggering event has to be regarded as illegal, is referred to as *rejection*.

ACTA: $e_1(t_1) \in H \Rightarrow e_2(t_2) \in H$

Production Rule:
ON $e_1(t_1)$
IF $\neg (e_2(t_2))$
DO reject(e_1, t_1)

Rule 3 Triggering Dependencies (Rejection)

Rule 3 shows the general scheme for the transformation of ACTA dependencies into production rule definitions. Here, the ACTA rule specifies the semantics of a dependency. The resulting production rule will fire upon occurrence of event $e_1(t_1)$ if the dependency has been established (see Rule 1 and Rule 2). Again, the event will be caught by the transaction broker, which also has a predefined reaction reject . The negated predicate in the condition part should not be confused with a negative event in the sense of active database systems [7].

If the appropriate reaction is to enforce the consequent instead of rejecting the triggering event, Rule 4 is applied. Similar to Rule 2, if the corresponding ACTA rule has a more complex consequent, the resulting production rule will have a (more complex) condition as well.

ACTA: $e_1(t_1) \in H \Rightarrow e_2(t_2) \in H$

Production Rule:
ON $e_1(t_1)$
DO raise($e_2(t_2)$)

Rule 4 Triggering Dependencies (Enforcement)

4.4 Realizing Restrictions on Operations Through Production Rules

In order to preserve correct histories involving operation events (in the sequel referred to as data manipulation events, or DML-events, for short) it has to be determined whether such an event is allowed at the current time (at the actual "position" in the current history). Similar to transaction dependencies, it is not acceptable to require that the history is represented explicitly and to reason about this explicit representation.

In order to evaluate the correctness of DML-events on behalf of a specific transaction, we have to check whether the object to be manipulated is in the view of the transaction. Furthermore, we have to check for conflicts with other transactions. Note that view sets are typically much simpler than conflict sets. For example, in the ACTA specification of Sagas [9], the view set of a Saga is empty (i.e., a Saga itself cannot perform any DML-operations), while the view set of a Saga component transaction is the entire current history (i.e., a component is allowed to access an object without restrictions). In both cases, checking the visibility of objects is very simple (always *false* in the first case, and always *true* in the second). For more elaborate cases, we assume that both the view set as well as the conflict set, are defined in terms of access sets of other transactions. Production rules will have to be generated for the maintenance of the access sets of the transactions, for looking up these sets in order to ensure correct accesses, and for registering conflicts.

The general transformation rule for DML-events is given in Rule 5 where p is a DML-event (name), t is a transaction, o is an operation and args optionally are arguments of the operation. The event validateView is caught by the scheduler (broker). Its reaction is to check whether the object is in the view set of the transaction. Similarly, validateConfl is caught by the scheduler which checks for conflicts with other transactions. Finally, while the reaction to registerAccess typically will be the update of the transaction's access set, execute is caught by the object manager, which executes the DML-operation as specified.

```
ACTA:               p_t (o, args)

Production Rule:    ON   p (t, o, args)
                    DO   raise (validateView(t, o))
                         raise (validateConfl(p, t, o))
                         raise (registerAccess(t, p, o))
                         raise (execute(t, p, o, args))
```

Rule 5 Production Rules for DML-events

For the understanding of the first three events it is necessary to recall that we are addressing the *construction* of a transaction management subsystem. Essentially, in this phase of construction we will try to reuse existing primitives that are responsible for the events (e.g., a lock manager for validation). One problem is to find primitives that obey the specified semantics. Given our approach to describing and implementing brokers by means of production rules, it is straightforward to describe the semantics of primitives (partially) with production rules as well. As an example, the validateView event would be described as in Rule 6. The meaning of the ACTA rule is to make solely those objects visible to transactions where either the object is currently not accessed by any transaction, or any transaction responsible for the object is also an ancestor of the requesting transaction.

ACTA: $View_t = \{p_{t'}(o) \mid (\neg\exists t': p_{t'}(o) \in \text{access-set}(t')) \lor (p_{t'}(o) \in \text{access-set}(t') \land \text{ancestor}(t, t')\}$

Production Rule:
ON validateView(t, o)
IF $\neg(\forall t_2: t_2 \in \text{accessingTAs}(o) \Rightarrow \text{ancestor}(t_2, t))$
DO reject(validateView)

Rule 6 Sample Production Rule for Validating View Set

The same approach is used for checking conflicts. Hereby, we assume that the conflict relation has been defined by the DBI. The resulting production rule defines the conflict check as the condition of the rule and also defines the reaction in case of conflict. We will next discuss additional strategies for the realization of conflict validation. First, some validation techniques (e.g., strict two-phase locking) would not simply allow or reject DML-events, instead they would try to re-order the current history attempting to preserve correctness. In this case, events or requests would be queued, and the scheduler made responsible for determining events to resume execution. This is supported by the operations defer and resume on events, which can be specified as reaction of production rules.

Furthermore, the simple transformation rule and the resulting production rule above (Rule 5) may need modifications in order to support the selection of components. Modifications are possible along the following lines:

- Additional events can be raised in the reaction part. For instance, *notification* events may be added in order to realize user notification upon the occurrence of specific events (e.g., changing design objects).
- The sequence of events raised in the reaction part can be modified, or some of these events can be assigned to other production rules. One criterion for classifying validation techniques is the *validation time*, the point in time DML-events are validated. For instance, if we would like to apply optimistic concurrency control, the validateConfl would not be raised in the reaction part of the DML-event, but would be assigned to a production rule triggered upon commit of responsible transactions.
- In order to optimize the execution of production rules, some events in the reaction part can be “merged”. Depending on the chosen technique, validation of DML-events and registration can be done in one step.

4.5 The Saga Example

As an example, we show the production rules realizing dependencies for the Saga transaction type. We conceive a Saga [15] (in the example represented by the variable S) as a sequence of component transactions t_i. A compensating transaction ct_i is attached to each component. The dependencies as given in the left-most column in Table 2 are then specified for Sagas, component transactions, and compensating transactions [9]. For the semantics of dependencies, refer to Table 1. The second column identifies the applied transformation rule. The three columns on the right side specify events, conditions, and actions of generated production rules, respectively. These dependencies are recorded by production rules generated according to Rule 1. Due to space limitations, we omit these production rules here. The first three dependencies are established at the begin of the Saga, the second three at the begin of the respective component, the next two when a component commits, and the last one at the begin of the last component in the sequence.

As for one special case, consider the seventh dependency. Here we would have to ensure that the compensating transaction ct_i commits if the Saga aborts (which is impossible to ensure, see the discussion above). Hence, we defined a production rule that fires in one incorrect case (namely, rollback, i.e., internal abort due to deadlock or something alike). In this case, the reaction is to restart the compensating transaction.

The last row contains another noteworthy example. This dependency specifies that the Saga has to commit if the last component commits as well. While we typically would refrain from enforcing commits and apply the rejection rule, enforcement is safe in this context. Hence, both examples show the power of the rule-based transfor-

Dependency	Transf. Rule	Event	Condition	Action
t_i BCD t_{i-1}	3	begin(t_i)	$\neg$committed(t_{i-1})	reject
ct_i WCD ct_{i+1}	3	begin(ct_i)	active(ct_{i+1})	reject
ct_{n-1} BAD S	3	begin(ct_{n-1})	$\neg$aborted(S)	reject
t_i AD S	4	abort(S)	-	abort(t_i)
t_i WD S	4	abort(S)	$\neg$ committed(t_i)	abort(t_i)
ct_i BCD t_i	3	begin(ct_i)	$\neg$ committed(t_i)	reject
ct_i CMD S	4	rollback(ct_i)	aborted(S)	restart(ct_i)
ct_i BAD t_i	3	begin(ct_i)	$\neg$ aborted(S)	reject
S SCD t_n	4	commit(t_n)	-	commit(S)

Table 2: Dependencies for Sagas and Corresponding Production Rules

mation approach.The first example shows the necessity to let the DBI edit transformation rules or specify new ones, while in some cases like the second example the DBI can choose whether to follow the enforcement or the rejection policy for select dependencies.

5 Conclusion

While the vast majority of work in the field of active DBMS elaborates on support of active mechanisms for concrete database systems (e.g., as an implementation mechanism for concepts like integrity enforcement [16, 17, 18], for the realization of control flow for long-running activities [19], for access control, as a mechanism to control heterogeneous system components [12], and so forth), our work uses production rules for two purposes:

1. as an internal, unifying, and integrating concept for the design and realization of DBMS,
2. as a construction paradigm for specific functionalities in the context of a DBMS construction method.

For the first purpose, we have introduced a broker/subsystem model for the construction of DBMSs, where the behaviour of brokers is defined in terms of production rules. Additionally, the production-rule-based approach provides good support for the parallel development of different subsystems, as well as providing for the specification of control flows between and within subsystems (brokers).

In comparison with the work done in the field of extensible database management systems [1], our approach bridges the gap between the analysis and design phase on one hand and realization on the other, whereas the known extensible DBMSs only address the issue of realization and leave more implementation work to the DBI.

Second, we have described an approach for realizing ACTA specifications of transaction models based on the broker/subsystem approach. This approach is transformational since it translates (logical) ACTA-rules into production rules. The transformation in turn is rule-based, resulting in a high degree of flexibility. Due to the rule-based transformation, we can cope with new dependencies that might be defined within the ACTA framework. A similar approach based on transformation of specifications into production rules is followed for the integrity enforcement subsystem [21].

One of our ultimate goals (developing a construction method in general and supporting the construction of transaction management subsystems in particular) is to support the choice among a broad variety of techniques for transaction management (mainly, concurrency control). To that end, we will have to investigate in greater detail how the broker-based approach can support the selection of artifacts from a (DBMS) software information base.

Acknowledgements

The work of A. Geppert has been supported by the Commission for the Support of Scientific Research (KWF), Switzerland.

References

1. Carey M, Haas L. Extensible Database Management Systems. ACM SIGMOD Record 19:4, 1990, pp 54-60.
2. Chrysanthis PK. ACTA, A Framework For Modeling And Reasoning About Extended Transactions. PhD Thesis, Technical Report 91-26, Department of Computer Science, University of Pittsburgh, November 1991.
3. Unland R, Schlageter G. A Transaction Manager Development Facility For Non Standard Database Systems. In: Elmagarmid AK (ed) Database Transaction Models For Advanced Applications. Morgan Kaufmann Publishers, 1992, pp 399-466.
4. Dittrich KR, Geppert A, Goebel V, Nittel S, Scherrer S. KIDS: A Declarative Approach to DBMS Construction. In: Spaccapietra S (ed) Proc. SI-Conf. on Database Research in Switzerland, Lausanne, September 1991, pp 176-185.
5. Biggerstaff TJ, Richter C. Reusability Framework, Assessment, and Directions. IEEE Software, July 1987, pp 41-49.
6. Dayal U. Active Database Management Systems. Proc. 3rd Intl. Conf. on Data and Knowledge Bases, Jerusalem, 1988.
7. Gatziu S, Geppert A, Dittrich KR. Integrating Active Mechanisms into an Object-Oriented Database System. In: Kanellakis PC, Schmidt JW (eds) Proc. 3rd Intl. Workshop on Database Programming Languages (DBPL), Nafplion, Greece, August 1991, Morgan Kaufmann Publishers, pp 399-415.
8. *The Common Object Request Broker: Architecture and Specification*. OMG Document 91.12.1, Revision 1.1, Object Management Group and X/Open, 1992.
9. Chrysanthis PK, Ramamritham K. ACTA: The Saga Continues. In: Elmagarmid AK (ed.) Database Transaction Models For Advanced Applications. Morgan Kaufmann Publishers, 1992, pp 346-397.
10. Pu C, Kaiser GE, Hutchinson N. Split-Transaction for Open-Ended Activities. Proc. Intl. Conf. on Very Large Data Bases, 1988, pp 26-37.
11. Bernstein PA, Hadzilacos V, Goodman N. Concurrency Control and Recovery in Database Systems. Addison-Wesley, 1987.
12. Buchmann A, Oezsu MT, Hornick M, Georgakopoulos D, Manola FA. A Transaction Model For Active Distributed Object Systems. In: Elmagarmid AK (ed.) Database Transaction Models For Advanced Applications. Morgan Kaufmann Publishers, 1992, pp 123-158.
13. Attie PC, Singh MP, Sheth A,Rusinkiewicz M. Specifying and Enforcing Intertask Dependencies. Proc. Intl. Conf. on Very Large Data Bases, August 1993.

14. Buchmann AP, Branding H, Kudrass T, Zimmermann J. REACH: A REal-Time, ACtive and Heterogeneous Mediator System. Bulletin of the IEEE TC on Data Engineering 15:1-4, 1992, pp 44-47.

15. Garcia-Molina H, Salem K. Sagas. Proc. Intl. ACM-SIGMOD Conf. on Management of Data, 1987, pp 249-259.

16. Ceri S, Widom J. Deriving Production Rules for Constraint Maintenance. Proc. Intl. Conf. on Very Large Data Bases, 1990, pp 566-577.

17. Eswaran KP, Gray JN, Lorie RA, Traiger IL. The Notions of Consistency and Predicate Locks in a Database System. Communications of the ACM 19:11, 1976, pp 624-633.

18. Kotz AM, Dittrich KR, Mülle JA. Supporting Semantic Rules by a Generalized Event/Trigger Mechanism. Proc. Intl. Conf. on Extending Database Technology, Venice, 1988, pp 76-91.

19. Dayal U, Hsu M, Ladin R. Organizing Long-Running Activities with Triggers and Transactions. Proc. Intl. ACM-Conf. on Management of Data, 1990, pp 204-214.

20. Branding H, Buchmann A, Kudrass T, Zimmermann J. Rules in an Open System: The REACH Rule System. First Intl. Workshop on Rules in Database Systems, Edinburgh, August/September 1993.

21. Scherrer S, Dittrich KR. Towards an Integrity Model Specification Language. Technical Report 93.13, Institut für Informatik, Universität Zürich, April 1993.