Exceptions in Information Systems

© Alfs Berztiss

Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260, USA

Bernhard Thalheim

Computer Science Institute, University of Kiel, Olshausenstrasse 40, 24098 Kiel, Germany
thalheim@is.informatik.uni-kiel.de

Abstract

The concept of exception has been defined in diverse ways. We relate exceptions to computational transactions and to control constructs. Our view of a transaction is very broad, and we consider transactional exceptions to be instances of undefined function values. By giving different interpretations to "undefined" we arrive at a classification of transactional exceptions. Our primary interest is in information systems, i.e., in database transactions, and in processes that consist of such transactions. In the database context we show that liberal treatment of exceptions is simpler than total quality management for consistency based on a set of constraints. We refer to control operations that link transactions into processes as actions. Actions tend to be time-related, and time Petri nets provide actions with semantics. The time Petri net representation indicates where exceptions can arise. We also consider high-level monitors for the detection of exceptions. Although our emphasis is on detection of exceptions, their handling is also discussed.

1 Introduction

An exception is some kind of deviation from the norm, and exceptions have been studied in the contexts of programming languages, information systems, and artificial intelligence. As can be expected, they have been defined differently in these areas. Despite their importance, generally they have not been given the attention they deserve. For example, in the 1719-page 2-volume Handbook of Software Engineering and Knowledge Engineering exceptions receive a very brief mention in just two places: p.126 of [1], and p.742 of [2].

We begin by stating a few definitions from the literature to show the variability between them:

1. An exception is an event occurring during execution of a program that makes continuation impossible or undesirable [3].
2. Exceptions are features that were added to programming languages to provide the programmer the capability to specify what would happen when unusual execution conditions occur, albeit infrequently [4].
3. An exception is inconsistency with the program specification [5].
4. We start by associating the occurrence of an exception with the violation of a constraint [6].
5. An exception is an unusual event, erroneous or not, that is detectable either by hardware or software and that may require special processing [7].
6. Exceptions are deviations from the ideal sequence of events [8].

From this list we can abstract out three general themes. First, the classical programming language approach (items 1 and 2): an exception handler aborts the program, or the program continues from the point of detection of the exception after a corrective action has been taken. Second, a violation of the software requirements is detected (items 3 and 4). Third, an exception is a deviation from normality (items 5 and 6). This may be an event or condition that prevents or delays the achievement of a goal that the user wishes to achieve. The goal-relatedness is not an essential characteristic of an exception, but an exception is required to be a relatively rare occurrence.

Another way of classifying exceptions is by their causes. Here four types can be distinguished. The first type is an error, which may relate to design, operation, or organization. A failure that results as a consequence of an error can be regarded as an exception, but it is preferable to consider it as a problem of quality management rather than of exception management. The second type is operational nondeterminism. In a lengthy numerical computation we cannot predict in advance if and when floating point underflow will occur, and, if it does occur, whether it will lead to division by zero. Even if a computation that is subject to operational nondeterminism is not terminated by an exception handler, the result of such a computation is unreliable. The best that an exception handler can then do is to give a warning to this effect. The third type is incompleteness. A software system operates in an environment (or context), which, following McCarthy [9], we denote by (w,t), where w is a slice of the world at time t. Unfortunately it is rarely possible to determine in advance all the components of w that are relevant, and how the relevant components are expected to

Proceedings of the 9th Russian Conference on Digital Libraries RCDL'2007, Pereslavl, Russia, 2007

evolve over time. It is impossible to determine in advance the effect of these components on a computation, which means that exceptions due to incompleteness require human intervention. The fourth type corresponds to the third theme listed above, namely deviation from normality.

Our purpose is threefold. First, in Section 2, we introduce a uniform definition of exceptions that encompasses all the classes introduced above. In this we distinguish between exceptions that relate to data and those that relate to control. Second, we put emphasis on exceptions that have received relatively little attention in the past. These are exceptions that arise in databases or in processes consisting of database transactions. They are discussed in Sections 3 and 4, respectively. Section 5 introduces a monitoring approach to the detection of unforeseen exceptional conditions. Section 6 deals with the design of exception handlers. Section 7 is a summary of our work and a look to the future.

2 Transactions and exceptions

2.1 Classification of transactional exceptions

One way of interpreting a computational process is to consider it as a sequence of transactions, which may be combined by control operations. We view a transaction as the evaluation of a function f for an argument or input x, where x and $f(x)$ can be single values or data aggregates of arbitrary complexity. The transactional view is natural for database operations, but it can be applied to any kind of computation. Thus $x := a$ is a transaction where ":=" represents the assignment function, a is its argument, and x is the value obtained. If we regard the data workspace of the program as a rudimentary kind of database, this differs little from a conventional database update. Transactions can be combined into composite transactions, i.e., transactions exist at different levels of granularity. For example, a program that generates the inverse of a matrix is a transaction made up of numerous primitive transactions.

This transactional–functional view is somewhat artificial, but it allows us to define in a uniform way all exceptions except those relating to control. It also leads to a classification. *An exception arises when f is undefined for a particular argument x*. This definition is very general, but different interpretations of "undefined" lead to different classes of exceptions.

(a) *Formal undefinedness*. Examples of this are division by zero, and square root of a negative number. The division function is undefined for a single value in its domain, the square root function for half the real numbers. The square root example shows that exceptions can relate to large subsets of possible arguments.

(b) *Processor-related problems.* Here the well-known examples are numerical overflow and floating point underflow. An arithmetic operation becomes undefined when the expected result is outside a processor-specific numerical range.

(c) *Inapplicable attributes.* This class if exceptions has been studied in the artificial intelligence context: "birds fly" does not apply to all birds. Here undefinedness relates to functions that have to do with flying, for, example, the top flying speed for a class of birds. Of course, the top speed can be set to zero for penguins, but a weight-to-speed ration for penguins is formally undefined.

(d) *Null values*. An inapplicable attribute is a type of null value, and null values can be represented in a database by special markers standing for inapplicable (as in our Class c), or knowable but not known (name of spouse of a married person), or unknowable (the names of all residents of a particular house in Pompeii in the morning of August 24, A.D. 79).

(e) *Database constraint violation*. Suppose that the total salary budget of a company is S. A salary increase is "undefined" if it were to result in salary expense in excess of S. Database constraints often act as filters. For example, if a transaction is to be applicable only to people at least 65 years old, then it is undefined for younger people, and applicability is defined by means of a constraint. This class of exceptions will be looked at in some detail in Section 3.

2.2 Transaction systems

We sketch a model of transactional computation in which a process is defined by a set of transactions and a set of actions that link the transactions into a process. Exceptions relate to both transactions and actions. Formally, a transaction system is the tuple $< T,A,S,F >$, where

T is a set of transactions,
A is a set of actions,
S is a set of signals,
F is a flow relation: $F \subset (T \times A) \cup (A \times T)$.

In a conventional program the flow relation is implicit: unless otherwise indicated, statements are executed in the sequence they have in the program.

The separation of process aspects from operations on data allows these components to be designed more or less independently of each other. Communication is by means of signals. Transactions send out signals, and signals are picked up by actions.

2.3 Control and exceptions

One characteristic of transaction systems is that they often exhibit a time dependence. The dependence can be of two kinds. One relates to data that can undergo dynamic change over time. This is a database management concern. The other relates to process control. Time-related exceptions include a deadline not being met by a single process, a transaction not taking place in a specified time window, and a rendezvous not achieved by several processes within a specified time

window. The deadline problem is also known as a timeout situation. For example, a telephone user has to complete dialing within a specified time interval.

An example of the time window problem arises with shunting of packages. As a package passes a bar code reader, its destination gate is determined, and the time of travel to the gate estimated — if the gate is opened too early, some other packages will enter the gate wrongly; if too late, the package has moved past the gate. In the rendezvous problem, suppose the first process arrives at time T. We require the other processes to arrive in the time interval $(T,T + t)$, where t is specified in the system requirements.

Another type of time dependence is temporary exception: a transaction is to be part of a process, or, alternatively, is to be bypassed, during a given time period. For example, Pennsylvania sales tax is in general applied to computers, but there have been periods of time in which they have been exempt from the tax. During this time the transaction that deals with the sales tax in not to be invoked.

A temporary exception defines two processes. In our example the processes are sale with and sale without sales tax. In other words, there is branching. In a more general sense we consider a process that consists of a normal sequence or network of transactions, and of exceptions. An example of a normal sequence is a process that supplies a customer with money from a banking machine. Exceptional situations arise with the use of a bank card that has been reported lost, withdrawal limits exceeded, machine out of cash, etc. In each instance there is branching. However, for this to be an exceptional situation, one of the branches is to be taken rarely. It is matter of subjective judgement whether or not a branching event is an exception. For example, the Pennsylvania sales tax exemption applies to *all* purchases of computers over a given time period. In this sense it is not a rare occurrence. On the other hand, the sales volume of computers, to which alone the temporary exemption applies, is small compared to the sales volume of items subject to sales tax.

Our coordination component is a collection of actions. Actions are started by signals received from transactions or sensors, or by clocks, or by people, or by some combination of the above, and they invoke transactions. We shall represent transaction systems by time Petri nets (for examples see [10], for a theoretical survey see [11]). Intuitively we expect a one-to-one correspondence between transactions and places, and between actions and transitions. This works for transactions, but not for actions. Because of the complexity of today's systems, arbitrary complexity must be allowed for in the specification of actions, although in most applications very little of this power is needed. An action is thus represented not by a single transition, but by a Petri net that can reach considerable complexity. Such a representation of actions is discussed in detail in Section 4.

3 Exceptions and databases

3.1 Occurrence of exceptions

Database systems are designed to be robust against errors. Error-handling facilities such as

- a transaction manager for management of concurrent access and computation on the database,
- a recovery manager for treatment of system, hardware, and software faults,
- an authorization and security manager for detection of intruders and security violation, and
- an exception programming environment for explicit specification of exceptions

are integrated into the system. Nevertheless, a number of other exceptions force the database developer and the database programmer to explicit treatment of deviations from the "normal" state.

Exceptions may occur due to

- modeling incompleteness caused by incomplete knowledge of the application domain and insufficient coverage of the concepts represented in the schema,
- insufficient implementation support for the database lifecycle, restrictions of modeling and programming languages, lack of attention by a modeler, and [12] a local instead of a global view of database constraints,
- pragmatic assumptions made during the database life cycle, hidden assumptions regarding what is the normal case in an application, overlooked cases, or a restricted scope of users,
- distributed computing of transactions under different commit protocols such as 2PC (problems of coordination, timeout of connections),
- internal organizational or computational restrictions of the database management system such as buffer management, memory restrictions, time restrictions etc.

It is usually assumed that a run of a database system fulfills the atomicity property, i.e. a transaction is either entirely successful and thus leads to a state change or is not having any effect on the data. Exceptions may, however, lead to unexpected behavior. Since each concept used in the database schema may have its own exceptional cases, the representation of all possible exceptions can lead to combinatorial explosion and to severe management problems. Thus, we need a mechanism that allows an "orthogonal" management of exceptions in the sense that each type has its own set of exceptions and an exception handler is called whenever an exception occurs. Such orthogonalization of exceptions is based on the introduction of general exception types. These general exception types may be instantiated by the exception handler in a restricted part

of the database. The instantiated exception programs may then handle the exceptions.

3.2 Specification of integrity constraints

We give particular attention to the development of general exception rules that are to support integrity constraints and show how integrity constraint enforcement is to be integrated with exception handling. As summarized in [13], the variety of static and dynamic integrity constraints is very large. Classically, database integrity constraints are specified as logical formulas. The logical framework provides a simple and powerful mechanism for treating the implication problem, for handling constraint sets, and for associating constraints with the database structures. But the framework neglects global effects. Also, if all possible exceptional cases are considered, the result in an overspecification. On the other hand, if only the "normal" case is considered, the result is underspecification. Moreover, normalization may introduce rigid constraint enforcement and thus leads to additional exceptions.

Database management systems do not support integrity constraints in full, restricting support for the most part to simple constraints such as primary key constraints, key dependencies, domain constraints, and referential inclusion dependencies. Functional or multivalued dependencies are not supported. Normalization of structures has been developed for treatment of functional or multivalued dependencies. SQL-99 provides more powerful database structuring mechanisms, but the treatment of constraints within this structuring framework has been an open problem [14].

The enforcement of integrity constraints is thus left to assertions, triggers, of stored procedures. Constraint enforcement thus becomes a difficult task. Trigger sets may lead to trigger avalanches or to non-intended effects. For instance, an insertion of a tuple may lead to deletion of all values of the tuple from the database. This behavior is based on the presence of critical paths. In [15] a sufficient and necessary condition for the existence of critical paths has been given. Effect preservation is far more difficult. It has been tackled in [16–19]. We list now a number of aspects of the constraint satisfaction problem that suggest why the treatment of constraints via exceptions can be very useful.

Rigidity of validity:
: Some integrity constraints are very important in an application area. Others are less important. Users can often "live" with temporary violations of the latter. Soft constraints [20] are constraints whose satisfaction is desirable, but not essential.

Behavior in presence of null values:
: Null values carry a variety of different semantics. Most constraints are not defined on null values. The behavior of some types of constraints such as functional dependencies becomes cumbersome if null values are permitted [21].

Exceptions of validity:
: In the daily operation of a database exceptions may arise due to various reasons. In some cases a constraint may be allowed to be invalid within a time interval. A validity exception may thus violate transaction semantics.

Enforcement time:
: Validity of constraints may be enforced at different points of time. This situation has been taken into account to some extent. For instance, SQL-99, allows one to specify that constraints are to be enforced whenever a tuple that might violate the constraint is modified, or at the end of the transaction, or based on the occurrence of some events. But the consistent management of constraint enforcement time is still an open problem.

Partial satisfaction of constraints:
: Constraints may be partially or totally satisfied [22]. We may collect all those tuples for which a constraint is not satisfied into a separate database unit.

Execution time deadlines:
: Constraints may be violated due to the late arrival of data or events. A contingency plan or contingency transactions may be invoked with guaranteed execution time characteristics.

Classically, integrity constraints are locally specified on the conceptual level without consideration of their enforcement and their scope within the schema. Constraint enforcement is added during the implementation phase. The environment of a constraint is formed by the associated types and by the effect of enforcement policies on other types and their constraints. SQL:1999 supports a number of strategies:

Direct enforcement
: can be automatically generated for declarative constraints on the basis of policy enhancements, such as `RESTRICT`, `NO ACTION`, `CASCADE`, `SET VALUE` (null, default), `[INITIALLY] IMMEDIATE [DEFERABLE]`.

Transactions
: provide three mechanisms for dealing with failure:
(1) rollback whenever an inconsistency is detected at the end of the transaction;
(2) advanced transaction models that erase the effects of a transaction by explicit specification of compensating transactions;
(3) DBMS support in the raising of an exception.

Triggers
: are procedures that may be automatically activated whenever an event occurs and a condition becomes valid. We may distinguish between integrity enforcement that depends on after-before activation time, on row-statement granularity, and on the possibility to use 1-n, n-1, or n-n event-trigger pairs.

The specification of integrity constraints should include their *environment* and the constraint *enforcement policy*.

Policies have been introduced in [17]; the environment has been discussed in [20]. Constraints are expressed as logical formulas. They can be restricted to a limited part of the database, called a unit. They can be partially violated. Exceptions for a constraint may be defined explicitly. The enforcement policy is specified by an enforcement rule with some kind of contingency framework. These principles are embedded in the following schematic logical formula or frame, which supports all the database exceptions discussed above:

Integrity Constraint ℘
[Localization: < unit_name >]
[Partiality: < validity condition >]
[Exception: < exception condition >]
[In-Context: < enforcement rule, time, granularity >]
[Out-Context: < conditional operation, accept_on >]
All components of this frame are optional.

3.3 Examples of database exceptions

For our example we take a small database in which are stored data on persons, on cars, and certain associations, e.g., that a car is owned by a person, where we let "own" to mean that the car has been purchased or is leased. Once a car is owned by somebody, insurance coverage must be obtained. A number of "exceptional" states may arise, brought about, for example, by the short interval within which insurance coverage has to be obtained or payments made to a dealer. In addition, cars may be returned to dealers, or may be scrapped. We could model all these specific cases by separate types, could use very general constraints to cover all exceptional cases, or use constraints that change dynamically depending on the state of objects. Such solutions lead to complex schemata that are difficult to understand, are not extensible, and have infeasible constraint maintenance. A better approach is to consider schemata with explicit specification of exceptions. We base our example in which this is done on the Entity-Relationship schema of Fig.1, which is expressed in the HERM notation (see [20]).

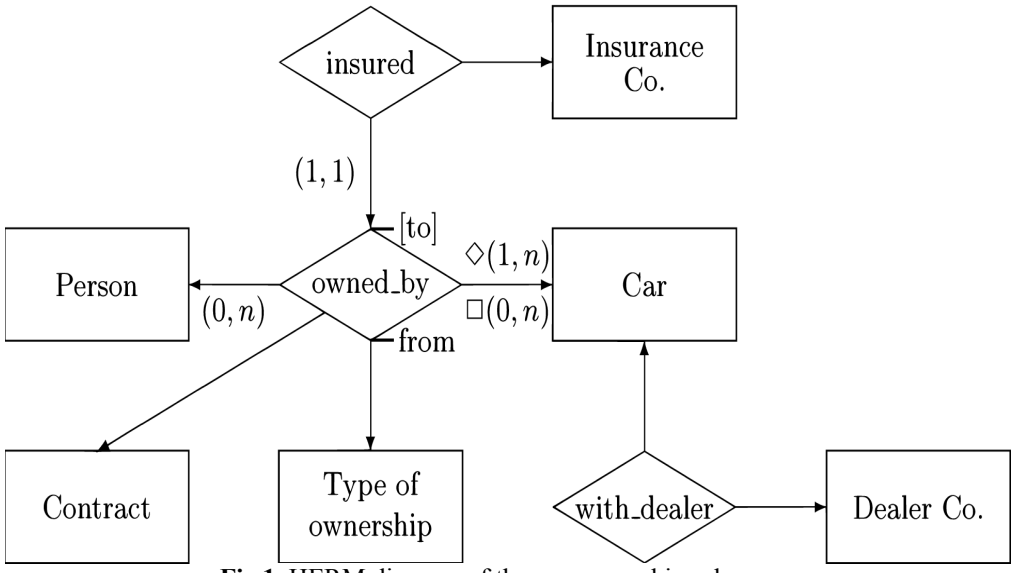

Fig.1. HERM diagram of the car ownership schema.

Whenever car data are inserted into the small database, the corresponding person data must already exist or must be inserted too. We may specify this constraint as follows:

Car_Must_Be_Owned_By_Person card(*Owned_By*, *Car*) = (1,*n*)
Localization: registration_department

The constraint can be equivalently expressed by the inclusion constraint
Car ⊆ *Owned_By*[*Car*].
We may, however, envision that a very small portion of cars stored in our database have no owners. An example is a car still waiting for a customer at a dealership. The constraint may be expressed as follows:

Car_Must_Be_Owned_By_Person card(*Owned_By*, *Car*) = (1,*n*)
Localization: registration_department
Partiality: if *Car* ⊆ *Cars_With_Dealer* ...
Exception: Unknown_Ownership

The constraint can be equivalently expressed by the inclusion constraint
$Car \setminus with\_Dealer[Car] \subseteq^{\theta} Owned\_By[Car]$.

be embedded into transaction management. We also may enforce integrity constraints at the row level or at the statement level. The combination of policies will lead to a global integrity constraint [13] if it can be

```
Car_Must_Be_Owned_By_Person card(Owned_By, Car) = (1,n)
Localization: registration_department
Partiality: if Car ⊆ Cars_With_Dealer ...
Policy:
```

On INSERT(x) Into *Car* If $x \not\in$ *owned_by*[*Car*] Do INSERT Into *owned_by* Immediately
On DELETE(x) From *Car* Do DELETE (.x) From *owned_by* Immediately
On UPDATE(x) On *Car* Do Cascade UPDATE On *owned_by* Immediately
On INSERT(x) Into *owned_by* If x.*Car* $\not\in$ *Car* Do No_Action Deferred
On DELETE(x) From *owned_by* If $card(\sigma_{Car=x.Car}(owned\_by)) = 1$ Do Restrict Immediately
On UPDATE(x) On *owned_by* Do No Action Immediately
On DELETE(x) From *Person* If $x \in$ *owned_by*[*Person*] Do No Action Deferred
On UPDATE(x) On *Person* If $x \in$ *owned_by*[*Person*] Do Cascade Deferred

We use the threshold value θ for expressing the size of the set of cars to which the exception may be applied.

The exception can be specialized as a set of policies: However, the specification as a set of policies tends to be error-prone and cumbersome, which further strengthens our view that constraint violation is best dealt with as exception management.

Furthermore, we may state that the beginning of an ownership of a car identifies the owner. This constraint is expressible by a functional dependency:

Owned_By : { Car, From } → Owned_By .

This constraint may be interrelated to the previous constraint:

```
Car_And_Date_Identify_Person Owned_By : { Car, From } → Owned_By
Localization: registration_department
Partiality: Unknown_Current_Ownership_Car_Again_With_Dealer
Exception: Unknown_From_Date
```

Functional dependencies are total constraints. We may also consider functional dependencies with null values. But this treatment also becomes very complex.

The example shows that explicit treatment of all exceptions may become a nightmare. Moreover, consistency of constraint enforcement policies is not axiomatizable and is not decidable [14]. Therefore, we must either restrict constraint enforcement to the "good" cases or leave constraint enforcement to the DBMS and the programmer. The total specification of all possible cases is already infeasible on the local level since the policy for an n-ary relationship type is specified by 3^{n+1} sub-policies. The deferred mode must

derived and computed. If we consider real life applications with a typical size of hundreds if not thousands of types, then treatment on the basis of policies becomes entirely infeasible. In this case the 'best' option is the pessimistic treatment of consistency, i.e., to entirely forbid any violation of consistency.

One goal of this paper is to show that we may weaken consistency enforcement by liberal constraint enforcement through acceptance of exceptions. This liberal treatment seems to be appropriate as long as we can specify a strategy that does not lead to an overwhelming volume of exceptions.

4 Exceptions and processes

4.1 Representation of actions

As noted earlier we view a computational process as composed of transactions and actions. Our experience has shown that textual representation of a complex action is easier to understand than the corresponding Petri net, but text is often ambiguous, and, as we discuss in detail in [24], this is the case with the specification language looked at here. Components of the language are therefore provided with standard interpretation in terms of time Petri nets. An example of an action:

```
ACTION Example;
  @(p) ON(SigA(a,b)) IN(0,q) EXCEPTION(ExA) ::
  DELAY(x,y) EXCEPTION(ExB):
  (TrA(a,b), TrB(b));
 ENDACTION
```

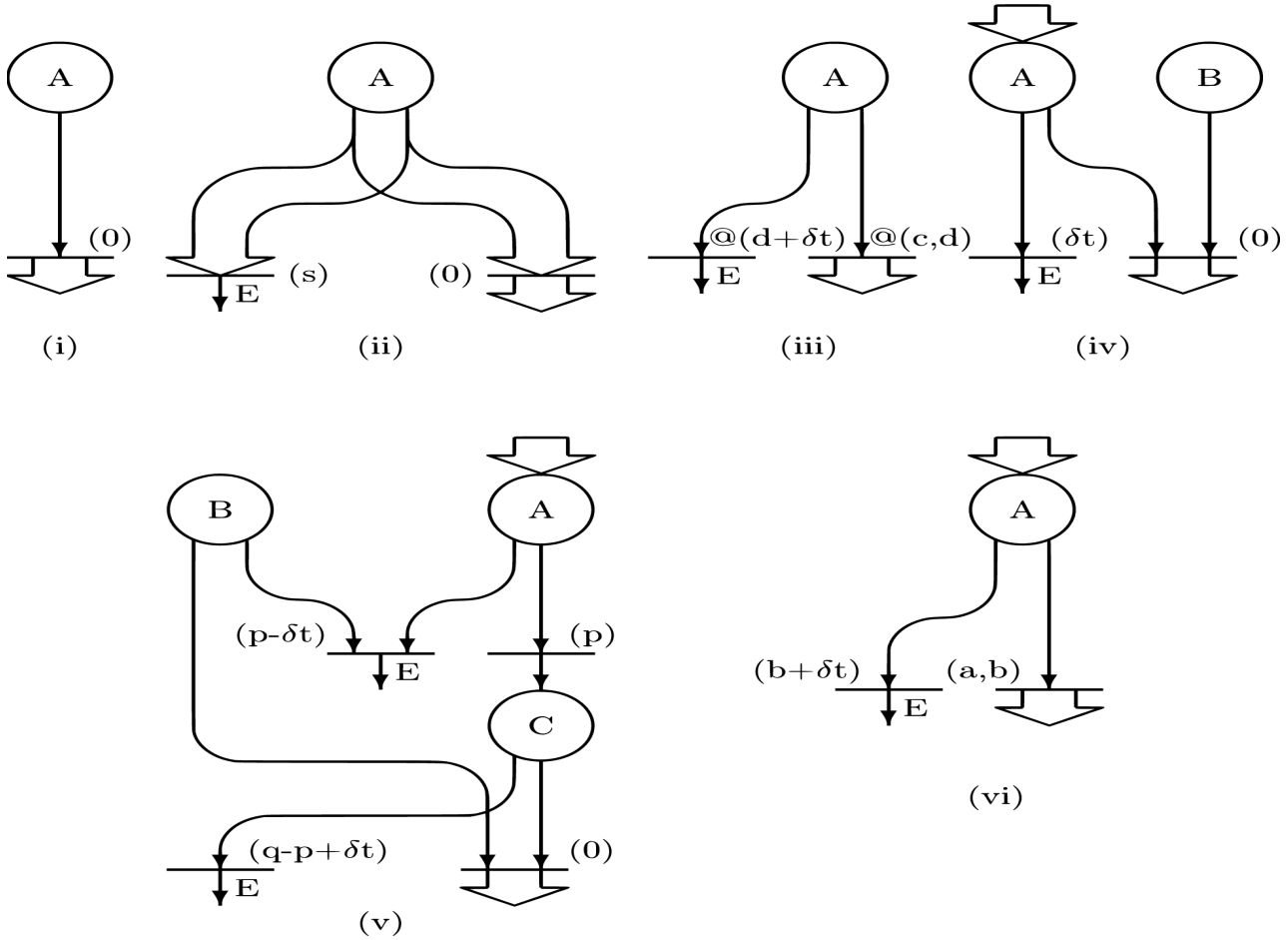

Fig.2. Components of actions

This action is initiated by a clock at time p. The action continues if signal SigA arrives no earlier than p and no later than p+q — this is what the IN component checks: unless the arrival of the signal is within the time interval (0,q) with respect to p, an exceptional condition arises, and the action terminates by invoking exception handler ExA. Otherwise, after a delay of between x and y time units, where the exact length of the delay is selected by an operator, transactions TrA and TrB are started. However, if the operator has neglected to initiate these transactions after y time units, exception handler ExB is invoked. This example shows a process controlled by both software and people. (In a simulation study the length of the delay would be determined by a random number generator.)

The syntax of the language for the specification of actions consists of seven productions expressed in BNF: Square brackets indicate that the item enclosed in the brackets is optional. If square brackets are followed by the symbol *, then the enclosed item may be present zero or more times; if followed by +, then the item must be present at least once. The symbol | indicates alternation, e.g., A ::= B|C indicates that A may be rewritten as B or as C. The example given above makes the syntax largely self-explanatory.

```
1. <Action>::          ACTION [<ActionId>];
                         <Activator>:: [<ActPart>;]*
                       ENDACTION
2. <Activator>::=      ON<Sig> | ON(<Sig> [,<Sig>]+)OFF <TPart> |
                         @<TPart> [ON(<Sig>) [<EPart>] |
                           ON(<Sig>)IN <TPart>]
3. <TPart>::=          (<TimeExp> [,<TimeExp>]) [<EPart>]
4. <Sig>::=            <SigId> [(<Exp> [,<Exp>]*)]
5. <EPart>::=          EXCEPTION(<PrimAct>)
6. <ActPart>::=        [<Delay>]
                         [<PrimAct> | (<PrimAct> [,<PrimAct>]*)]+
7. <Delay>::=          DELAY <TPart> [<EPart>]:
```

The form of identifiers and expressions (ActionId, TimeExp, SigId, Exp) is left undefined. The PrimAct

stands for a transaction, which may be an exception handler, or the activation of a mechanical device, or a message sent to a human operator.

The only component not used in the example is OFF. It becomes necessary when several conjoined signals are required, as in ON(SigA,SigB)OFF(15). This is a synchronization mechanism: the action does not advance until all the signals have been raised. But if a signal does not get raised in the time period in which it is expected to be raised, the system freezes. To avoid this, the mandatory OFF is provided. After a time interval *s* (here *s* = 15 time units), measured from when the first signal in the set is raised, all signals are switched off, and an exception handler is invoked. Here as everywhere else, the exception handler may decide to take corrective steps or to abort the action. Exceptions may be handled by computer programs or by people.

We have modified time Petri nets by allowing clock readings to specify the time an action is to be initiated. In terms of the syntax, a TPart preceded by the symbol @ represents clock readings; otherwise the TPart has its conventional interpretation. Fig.2 provides a standard interpretation of the components of an action. Every action can be represented by a time Petri net composed of the subnets of Fig.2. A broad arrow represents one or ore arcs.

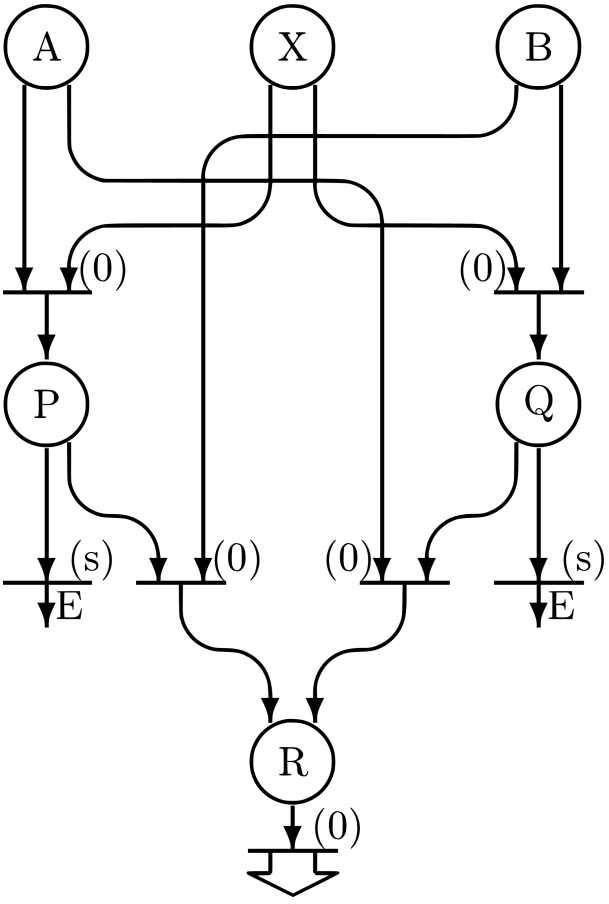

Fig.3. Conjunction of two signals

Case (i) corresponds to the first alternative of Production 2 — a signal starts an action. In Case (ii) several signals initiate an action, and this corresponds to the second alternative of Production 2. Case (ii) is complex, and its representation is merely schematic. The form of the net depends on the number of signals that are to be conjoined. In Fig.3 we show the detailed net for two conjoined signals. In this net, place X is initially to hold a token. Suppose that a signal is raised first by the transaction represented by A. Then the transition that leads to P fires, removing the token from X, and we wait for the signal from B. If it arrives in time, i.e., before *s* time units are up, the action continues. If not, the exception transition fires. If the signal from B arrives first, the situation is symmetrically analogous.

In Case (iii) the action is initiated by a clock or a person. If initiated by a clock, the setting would normally be $c = d$, but in a simulation study a random time within the interval (c,d) could be selected. If initiated by a person, a time period defined by two different clock readings would be usual. If the action has not been started at time d, an exception arises. Cases (iv) and (v) interpret the two optional components that can follow a clock-based initiation. In Case (iv) a signal has to be on at the time of initiation of the action — this signal is issued by a transaction represented by place B. If the signal is not on, we have an exception. In Case (v) we also require a signal to be on after the clock-based initiation, but not immediately. Rather, it should come on at a time within $(T + p, T + q)$, where T is the time at which the action is initiated. Exceptional situations arise if the signal is already on at $T + p$, or has failed to come on by $T + q$. An example arises with package routing by means of destination gates. Suppose a bar code reader selects a gate for a package. The bar code reader also initiates the action. The gate should not open too early or too late. Here the signal is issued by the gate-opening mechanism at the time it opens the gate.

In the remaining case there is to be a delay of between a and b time units, and, as we noted earlier, a manager determines the precise length of the delay. An exception arises if the manager fails to resume the action before the delay time is exceeded.

Fig.4 shows the Petri net corresponding to our example of an action. It is built up from the components of Fig.2 in a mechanical fashion. We leave it to the reader to identify the places and transitions that correspond to Cases (iii), (v), and (vi) of Fig.2. The author finds the specification of the action as text easier to follow than the net of Fig.4, but the net removes interpretation ambiguity.

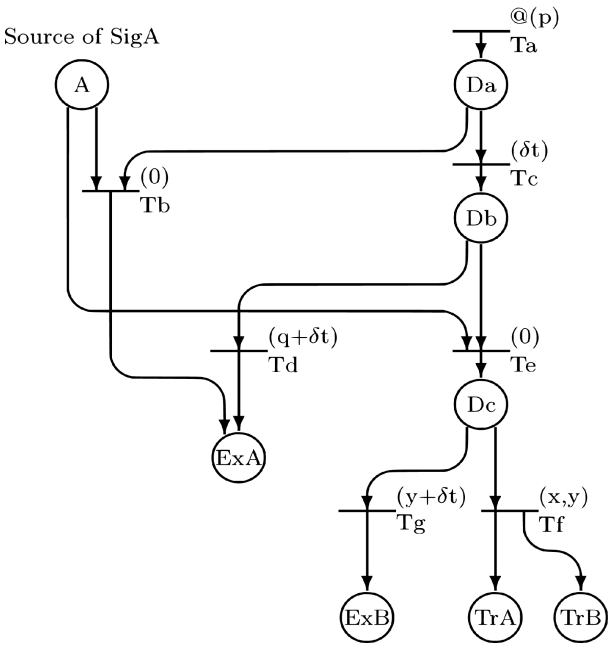

Fig.4. The net of an action

4.2 Example of a process

In Section 3 we looked at exceptions in the context of a database relating to ownership of cars. Here the example is extended into a process that a car undergoes from its manufacture to its final disposal. Fig.5 shows the process participants and lines linking them. Directed lines represent transactions that transfer a car from one participant to another, and we have attached a label to each such line.

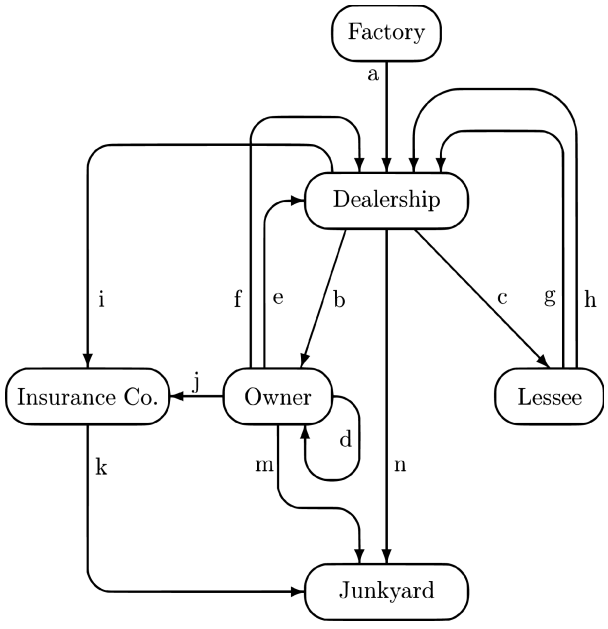

Fig.5. The lifetime process of a car

The initial transfer of the car is from factory to a dealership (Label a). There the car can be sold or leased (b and c, respectively). An owner may resell the car privately to another entity of type "Owner" (d) or to a dealership (e). The car may be repossessed by a dealership if the car was bought on terms, and no payments are made (f), and similarly for a leased car (g). At the end of the lease period the car is to be returned to the dealership, but under special circumstances an earlier return may take place (Label h). One cause for early return is damage to the car. If a car is damaged beyond repair, the insurance company takes over what is left of the car (Label i if the transfer is from a dealership; Label j if from an owner), and scraps the car (k). A car may also be scrapped by an owner (m) or a dealership (n).

Some of the lines in Fig.5 do not represent single transactions. For, example, under h, we have to distinguish between normal and early return, and in the latter case, between a total-loss collision, in which case h is at once followed by i, and an early return arising from lease cancellation by a lessee. Although repossession is also an early return, a separate line (line g) is necessary because this return is initiated by a dealership.

Fig.5 represents a distributed process. The transactions relate to different databases, maintained by dealerships and insurance companies. An additional process participant is the motorcar registration office with its database. We do not show this participant explicitly because it is not itself in possession of cars.

Now let us look at some exceptions that can arise. They all involve a time constraint. A dealership arranges temporal registration and insurance for a car when it sells it, but the new owner has to arrange for permanent insurance and permanent registration within a specified time period. Similarly, in an owner-to-owner sale, arrangements have to be made for insurance coverage and registration transfer within a time limit. For example, in

```
ACTION Insurance;
  ON Sig(CarX) ::
  DELAY(0, Limit) EXCEPTION(HandlerA):
  Insure(CarX);
 ENDACTION
```

the exception handler *HandlerA* is invoked if the action part that follows the DELAY has not been started *Limit* time units after signal *Sig* has initiated the action. Similarly, after a repossession, registration and insurance have to be taken over by the dealership within a specified time period.

Normally a leased car is to be returned at the end of the lease period. Here two exceptional conditions can arise: the car is returned before the end of this period, or it is not returned at the end of this period. Both cases are covered by

```
ACTION LeaseReturn;
  ON Leased(CarY) ::
  DELAY(0, Leaseperiod) EXCEPTION(Late):
  Return(CarY);
 ENDACTION
```

If the car is returned before the end of the lease period, then this is taken into account by the transaction *Return*, which is invoked by an operator; if there has not been a return at the end of the lease period, exception handler *Late* is invoked. Note that early return may be due to an accident. Action *LeaseReturn* raises questions. First, if there is an early return, why not invoke *Return* directly, instead of taking our roundabout approach? The reason is that the action has to be terminated. Otherwise, after the end of the normal lease period, although the car had already been returned, exception *Late* will be wrongly raised. Second, what if an exceptional situation arises within the action itself? A lease may cover several years. During this period the code of *LeaseReturn* may get changed, or a move of the system to another platform may cause the clock-based trigger that should raise the exception to malfunction. This can be handled by monitors, which we discuss in Section 5.

The main problem of exception management is the determination of what to do when an exception has not been anticipated in system design. In our example we have not considered what happens when a car gets stolen, or is confiscated because of its involvement in a crime, or is impounded because of failure to pay parking fees. In the latter two cases an authority may sell the car. Here we are becoming aware of the oversights still during the design phase, but it could well happen that a system is made operational with these flaws.

Let us now consider a situation in which a customer requests from a dealership a car with a specific attribute set *Q*. The request is forwarded to the factory, and several possibilities can arise. First, the specified car is available for immediate delivery. This is the normal situation, covered by the transfer from factory to dealership to owner shown in Fig.5 as lines a and b. Second, the car can be made available, but after a delay. Two exceptional situations can arise: (a) the customer finds the delay unacceptable; (b) the factory may not be able to supply the car within the initially suggested delay period. Another type of exception arises when the factory does not respond to the initial request within a reasonable time period. Except when an order can be filled at once, this special-order process is separate from that of Fig.5 because the car being considered does not yet exist.

4.3 Exception patterns

Both the example actions shown above have the same pattern, which we express in terms of the irreducible components of our grammar:

```
ACTION <ActionId>;
  ON <Sig> ::
  DELAY(<TimeExp>, <TimeExp>)
  EXCEPTION(<PrimAct>):
  <PrimAct>;
 ENDACTION
```

Actually the pattern embodies Case (vi) of Fig.2. Cases (ii), (iii), (iv), (v), and (vi) can be regarded as generic exception patterns for processes. Fig.6 illustrates an instance of Case (ii) of Fig.2. Transactions TraX and TraY send out signals SigA and SigB. In addition, SigC and SigD are sent out by some other transactions. Action Act1 is to be initiated by a conjunction of signals SigA, SigB, SigC, and Act2 by a conjunction of SigA, SigB, SigD. Now, if Act1 picks up SigA and SigC, and Act2 picks up SigB and SigD, both actions go into a wait state — a deadlock situation has arisen in which Act1 waits for the now unavailable SigB, and Act2 waits for SigA. Deadlock can be prevented by by a mutual exclusion mechanism, but a simpler solution is to invoke an exception handler that allows, say, both actions to proceed even though each of them lacks one of the required signals.

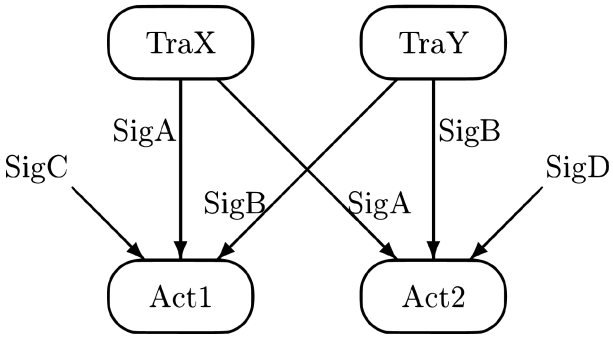

Fig.6. A deadlock situation

Thus there are essentially just five basic exception types associated with actions. In addition there is the case in which a signal fails to be issued over a suspiciously long time period. This small number appears to be in conflict with the 260 exception types collected by the MIT Process Handbook Project [25], but we should note that many of the latter are essentially design errors, such as " goals contain conflicts or inconsistencies," or "process contains design flaws" [8]. Our contribution has been to show that there there is a very small number of basic components of an action, and to make exception detection a precisely defined part of these components.

5 Monitor-based exception detection

In Case (i) of Fig.2 an action is to be initiated by a signal. No link to an exception handler is provided because the absence of a signal need not imply an exceptional situation. However, if an action is to be initiated by a signal, then we do expect this signal to be raised eventually. A similar situation can arise with Case (ii). Here an exception is detected when some signals do not arrive in time, but no provision is made for a case in which the system has to wait for the first of the signals an inappropriately long time. We do not want to impose time constraints on the signals, primarily because they could differ for different instances of an application, but we still need to make sure that the absence of signals is not due to communication failure or some other anomaly. The detection of anomalies can be handled by monitors. Exception monitors serve several purposes.

First, they act as a safeguard in case exception detectors and exception handlers are faulty. A problem with handlers is that they may not be adequately tested. If test case selection is based entirely on an operational profile (on operational profiles see, for example, [26]), then exception handlers, because they are rarely invoked, may not get tested at all.

Second, monitors should be made responsible for detecting and handling exceptional situations that have not been handled in system requirements. In a disciplined process for the development of information systems, requirements should be a refinement of the goals that an enterprise has set for the information system. It has been suggested that an exception monitor should be based on such goals [27]. However, if monitors are to be goal-based, why not make the requirements a complete refinement of the goals? This shows that we are dealing with a difficult research topic. We leave it for the future.

Third, monitors should be a safeguard against communication breakdowns. In case a signal does not initiate an action when expected, there can be multiple causes that we examine in the next section. One such cause is the failure of a communication link. A monitor can survey the situation and establish an alternative communication path.

6 Exception handlers

An exception detector merely established that an exception has arisen, and invokes an exception handler. A major aim of the developers of information systems is to implement exception handlers as software. Here we look in general terms at the structure of a software exception handler.

The first step is to establish the precise cause for the exception. This also applies to exception handling carried out by humans. For example, if an expected signal has not arrived from transaction *X*, there can be the following causes:

- communication failure between *X* and the action;
- transaction *X* was not initiated;
- transaction *X* has failed;
- a partial redesign of the process has resulted in turning the action into dead code.

Next the selected cause has to be examined further. For example, if transaction *X* was not initiated, then the exception relates not to the action, but to transaction *X*. The reason why *X* was not initiated could be that a human operator was not available to initiate it, or that a different transaction was initiated in error. The failure may sometimes have to be traced back through a chain of transactions and actions until the actual reason for the exception has been established. Only then can corrective action be undertaken. As part of this there should be an estimate of the actual damage, monetary or as reduced goodwill, that the exceptional situation has caused.

7 Summary and a look to the future

We have considered exceptions in two contexts: as they relate to database transactions, and as they can arise in process execution. With regard to database transactions, we demonstrated how different kinds of exceptions may be used for handling violations of integrity constraints. The exception handler calls a program that moves the database into a state that is consistent with the specification. A temporal violation of integrity constraints is allowed as long as the exception handler allows the transaction to reach a consistent state. With regard to processes, we have made explicit the detection of exceptional situations in our language of actions, which in an earlier form was introduced in [24].

The very general definition of transactions in Section 2 suggests that the “action-language” of Section 4.1 can be added as a coordination component to any modular specification language that needs only to be extended to allow for the sending out of signals, or that already possesses such capability. The transformation of the action specifications into executable code should not present difficulties. For ease of implementation, signals should then be directed to specific actions, which implies that the naming of actions becomes mandatory.

In Section 4.3 we saw that actions follow general patterns that are based on the components of actions of Fig.2. This enables us to pinpoint precisely the point from which a backward trace through the system is to lead to the actual cause of the exception. Sometimes the response to an exception has to take place so rapidly that human response times are inadequate. The precise localization of the manifestation of exceptions should help toward the automation of exception handling. Unfortunately this is not enough. We saw that although the number of basic types of exceptions is small, there can be a variety of causes for an instance of an exception. Consequently, cause-effect relationships have to be examined very thoroughly, and the cause-effect analysis automated to the greatest extent possible.

In Section 5 a brief introduction was made to exception detection my monitors. The design of monitors, or, rather, the establishment of general principles to guide their design, we consider a very important research topic. If an exception manifests itself to the system merely as a failure, then the search for its cause can be laborious and difficult. Monitors can be designed in such a way that they detect causes directly.

Bibliography

[1] Chang, S.-K. (Ed.), Handbook of Software Engineering and Knowledge engineering, Volume 1. World Scientific, 2001.

[2] Chang, S.-K. (Ed.), Handbook of Software Engineering and Knowledge engineering, Volume 2. World Scientific, 2002.

[3] Maxion, R.A., and Olszewski, R.T., Eliminating exception handling errors with dependability cases: a comparative, empirical study. IEEE Trans. Software Eng. 26 (2000) 888–906.

[4] Ryder, B.G., and Soffa, M.L., Influences on the design of exception handling. Software Engineering Notes, Vol.28, No.4 (July, 2003) 29–35.
[5] Romanovsky, A., Xu, J., and Randell, B., Exception handling and resolution in distributed object-oriented systems. In Proc. 16th IEEE Internat. Conf. on Distributed Object-Oriented Systems, IEEE CS Press 1996, 545–552.
[6] Murata, T., and Borgida, A., Handling of irregularities in human centered systems: a unified framework for data and processes. IEEE Trans. Software Eng. 26 (2000) 959–977.
[7] Hagen, C., and Alonso, G., Exception handling in workflow management systems. IEEE Trans. Software Eng. 26 (2000) 943–958.
[8] Dellarocas, C., and Klein, M., A knowledge-based approach for handling exceptions in business processes. Information Technology and Management 1 (2000) 155–169.
[9] McCarthy, J., Notes on formalizing context. In Proc. 13th Internat. Joint Conf. Artificial Intelligence, 1993, 555–560.
[10] Berztiss, A.T., Petri nets. To appear in Handbook of Software Engineering and Knowledge engineering, Volume 3. World Scientific.
[11] Berthomieu, B., and Diaz, M., Modeling and verification of time dependent systems using time Petri nets. IEEE Trans. Software Eng. 17 (1991) 259–273.
[12] Berztiss, A.T., Should integrity constraints be global or local? Sixth Internat. Workshop on Foundations of Models and Languages for Data and Objects, Schloss Dagstuhl, Germany, 1996.
[13] Thalheim, B., Dependencies in Relational Databases. Teubner Verlag, 1991.
[14] Schewe, K.-D., Design theory for advanced datamodels. In Proc. ADC 2001, ACM Press, 2001, 3–9.
[15] Schewe, K.-D, and Thalheim, B., Limitations of rule triggering systems for integrity maintenance in the context of transition specification. Acta Cybernetica 13 (1998) 277–304.
[16] Balaban, M., and Jurk, S., A DT/RT/CT framework for integrity enforcement based on dependency graphs. In Proc. DEXA 2001, LNCS 2113, Springer, 2001, 501–516.
[17] Balaban, M., and Jurk, S., Effect preservation as a means for achieving update consistency. In Proc. FQAS 2002, LNCS 2522, Springer, 2002, 28–43.
[18] Link, S., and Schewe, K.-D., An arithmetic theory of consistency enforcement. Acta Cybernetica 15 (2002) 379–416.
[19] Link, S., Consistency enforcement in databases. In Proc. Semantics in Databases, LNCS 2582, Springer, 2003, 139–159.
[20] Thalheim, B., Entity-Relationship Modeling – Foundations of Database Technology. Springer, 2000.
[21] Levene, M., and Loizou, G., A guided tour of relational databases and beyond. Springer, 1999.
[22] Buchmann, A.P., Carrera, R.S., and Vazquez-Galindo, M.A., A generalized constraint and exception handler for an object-oriented CAD-DBMS. IEEE Conf. on OODBS, 1986, 38–49.
[23] Balaban, M., and Shoval, P., MEER - An EER model enhanced with structure methods. Information Systems 27 (2002) 245–275.
[24] Berztiss, A.T., Time in modeling. In Information Modelling and Knowledge Bases XIII. IOS Press, 2002, 184–200.
[25] Malone, T.W., Crowston, K., and Herman, G.A. (Eds.), Organizing Business Knowledge — The MIT Process Handbook. MIT Press, 2003.
[26] Musa, J.D., Operational profiles in software-reliability engineering. IEEE Software, Vol.10, No.2 (March, 1993) 14–32.
[27] Lamsweerde, A.v., and Letier, E., Handling obstacles in goal-oriented requirements engineering. IEEE Trans. Software Eng. 26 (2000) 978–1005.2 Format, style and content