Generation of Database Transactions with Petri Nets

Kees M. van Hee, Natalia Sidorova, Marc Voorhoeve, and Jan Martijn van der Werf

Department of Mathematics and Computer Science
Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{ k.m.v.hee, n.sidorova, m.voorhoeve, j.m.e.m.v.d.werf }@tue.nl

Abstract. In this paper we show how we can generate models for the execution of database transactions. We describe a transaction goal by a data model and we give algorithms to generate Petri nets models that specify the execution of these transactions. This is done in such a way that database constraints, in particular referential integrity constraints, are preserved. So if the database is in a consistent state before a transaction starts, it will be in a consistent state after the transaction. The class of Petri nets we use is a subclass of coloured Petri nets, where token values are vectors of identifiers. This class is powerful enough to model transaction execution and it allows for some formal analysis, like soundness.

1 Introduction

Database theory is mainly focused on relational algebra, [6], query processing [12] and on correct handling of concurrent processing of multiple transactions [7]. On the other hand, database modeling focusses on modeling entities and their relations. However, the design of individual complex database transactions is mainly considered as a programmer's task, as the authors of [12] state: *it is the programmer's responsibility to define properly the various transactions, so that each preserves the consistency of the database*, i.e. the programmer has to ensure that his transactions preserve consistency of the database.

Database languages like SQL do not guarantee the consistency of the database. Operations like insert and delete only work on single tables. As soon as multiple entities of different tables have to be altered, transactions (i.e. a sequence of SQL queries) are needed. Techniques like stored procedures and triggers assist the programmer but do not guarantee consistency of the database. Currently, the main technique for guaranteeing cardinality constraints, and implicitly, referential integrity, is the construction of a table structure such that each insertion and deletion of records in tables preserves the referential integrity [2, 3, 5, 6]. In this paper, we show a completely different approach where data models with cardinality constraints are used to generate processes for different types of transactions that preserve the constraints, i.e. if the database is consistent before a

transaction, it is consistent afterwards. The advantage of this approach is that only the basis of the transaction is described by the programmer, and the rest is automatically generated ensuring the referential integrity. Although we only illustrate the method for referential integrity, we expect the method applies to other kinds of constraints as well.

In this paper we use a subclass of Petri nets to model database transactions. Classical Petri nets [9] with valueless tokens are a powerful tool to model complex control flow problems, but when data plays an essential role we have to use coloured Petri nets [8]. Coloured Petri nets are very powerful for modeling, but the analysis techniques are very limited [8]. In many data intensive applications, only objects and their identities play an essential role. In order to deal with these kinds of applications, we need limited operations such as equality testing on identifiers and copying of identifiers. In this paper, we introduce a class of coloured Petri nets where tokens are vectors of identifiers, and there is a restricted set of operations on transitions: only testing for equality on identifiers and copying of identifiers. On the one hand verification of behavioral properties for this class of coloured Petri nets is easier [10] and on the other hand the class is powerful enough to model database transactions.

2 Preliminaries

Let S be a set. $|S|$ denotes the number of elements in S. The *powerset* of S is denoted by $\mathcal{P}(S) = \{S' | S' \subseteq S\}$. A *bag* (*multiset*) m over S is a function $m\colon S \to \mathbb{N}$, where $\mathbb{N} = \{0, 1, \ldots\}$ denotes the set of natural numbers. The set of all bags over S is denoted by $\mathbb{B}(S)$. We identify a bag with all elements occurring only once with the set containing the elements of the bag, and vice versa. We use $+$ and $-$ for the elementwise sum and difference of two bags, and $=, <, >, \leq, \geq$ for the comparison of two bags, which are defined in the standard way. We use $\emptyset$ for the empty bag, and $\in$ for the element inclusion.

The cartesian product of two sets A and B is defined as the set $A \times B = \{(a, b) \mid a \in A, b \in B\}$. The generalized cartesian product for a set I, and sets A_i, $i \in I$, is defined as

$$\prod_{i \in I} A_i = \{f\colon I \to \bigcup_{i \in I} A_i \mid \forall i \in I\colon f(i) \in A_i\}.$$

An element $x \in \prod_{i \in I} A_i$ is called a *vector*. We define $\pi_i(x) = x(i)$, for $i \in I$. the definition of π_i is lifted in a standard way to sets: $\pi_i(B) = \{\pi_i(b) | b \in B\}$. If $|I| = n$ and $A_i = A$ for all $i \in I$, we write A^n for $\prod_{i \in I} A_i$.

A Petri net is a 3-tuple $N = (P, T, F)$ where (1) P and T are two disjoint sets of *places* and *transitions* respectively; (2) $F \subseteq (P \times T) \cup (T \times P)$ is a *flow relation*. We call the elements of the set $P \cup T$ nodes of N, elements of F are called *arcs*. Places are depicted as circles, transitions as squares. for each element $(n_1, n_2) \in F$, an arc is drawn from n_1 to n_2.

A Petri net $N' = (P', T', F')$ is a subnet of a Petri net $N = (P, T, F)$, denoted $N' \subseteq N$ if and only if $P' \subseteq P$, $T' \subseteq T$ and $F' = F \cap ((P' \times T') \cup (T' \times P'))$.

Let $N = (P, T, F)$ be a Petri net. Given a node $n \in P \cup T$, we define its preset ${}^\bullet_N n = \{n' | (n', n) \in F\}$, and its postset $n^\bullet_N = \{n' | (n, n') \in F\}$. If the context is clear, we omit the N in the subscript.

Markings are states of a net. A *marking* m of N is defined as a bag over P. A pair (N, m) is called a *marked Petri net*. A transition $t \in T$ is *enabled* in a marking $m \in \mathbb{B}(P)$, denoted by $(N, m)[t\rangle$ if and only if ${}^\bullet t \leq m$. Enabled transitions may *fire*. A transition firing results in a new marking m' with $m' = m - {}^\bullet t + t^\bullet$, denoted by $(N, m)\ [t\rangle\ (N, m')$.

3 Petri Nets with Identifiers

In classical Petri nets, we only deal with control flow, tokens are therefore valueless. In order to model database transactions, we need data objects, modeled using tokens with identifiers.

Therefore, we present an extension of Petri nets, where tokens have a simple colour, which is a vector of identifiers. By working with vectors, instead of primitive identifiers, we are able to combine objects. Since an identifier can refer to a set of other identifiers, we need to be able to test for the empty set. Therefore, we also define inhibitor arcs on identifiers. The presented extension is a natural combination of the class of Mobile Synchronizing Petri nets(MSPN) [10, 11] and inhibitor nets. In our extension, each token is a vector of identifiers. A vector of length 0 represents a black token. Each arc has a vector of variables, and the length of this vector is defined by the length of the vector of the place connected by the arc. The class of MSPN is a subclass of our proposed extension, by allowing only vectors of length 0 or 1.

Let the countably infinite set $\mathcal{I}$ denote the universe of identifiers, and let Σ be a non-empty set of variables. A transition that can create a new identifier is called an *emitter*.

Definition 1 (Inhibitor Petri net with identifiers). *An inhibitor Petri net with identifiers (PNID) N is a 7-tuple $(P, T, F, H, \alpha, \beta, \gamma)$, where*

1. *(P, T, F) is a Petri net; $H \subseteq (P \times T)$ is the set of inhibitor arcs. For a transition $t \in T$, the set of places tested for zero is defined as $H(t) = \{p \in P | (p, t) \in H\}$;*
2. *$\alpha \colon P \to \mathbb{N}$ defines for each place the length of the vector of identifiers in the tokens residing at the place. The color type C of each place $p \in P$ is defined as $C(p) = \mathcal{I}^{\alpha(p)}$. If $\alpha(p) = 0$, the place holds black – uncoloured – tokens;*
3. *β is an inscription function over the set of arcs: $\beta \in \prod_{f \in F} V_f$, where $V_{(p,t)} = V_{(t,p)} = \Sigma^{\alpha(p)}$ for $p \in P, t \in T$.*
4. *γ is an inscription function over the set of inhibitor arcs: $\gamma \in \prod_{(p,t) \in H} \Sigma^{\alpha(p)}$.*
5. *for a transition $t \in T$, we define its set of input variables $I(t) = \{v | v \in rng(\beta(p, t)), p \in P\}$, the set of output variables $O(t) = \{v | v \in rng(\beta(t, p)), p \in P\}$, the set of creator variables $\mathrm{new}(t) = O(t) \setminus I(t)$, and the set of variables $\mathrm{var}(t) = I(t) \cup O(t)$. $E = \{t \in T | new(t) \neq \emptyset\}$ defines the set of emitters.*

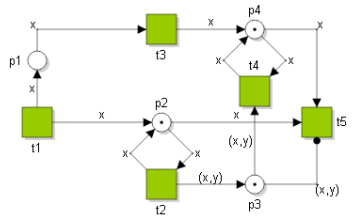


Fig. 1. An example of an inhibitor Petri net with identifiers.

An arc $f \in F$ is annotated with inscription $\beta(f)$. If $\beta(f)$ is the empty vector or a vector of length 1, we omit the brackets. Figure 1 shows an example of a Petri net with identifiers. In this example, we have two emitters, t_1 and t_2, and $\alpha(p_1) = \alpha(p_2) = \alpha(p_4) = \alpha(c) = 1$, and $\alpha(p_3) = 2$.

A marking of a Petri net with identifiers specifies the locations of tokens with vectors of identifiers in the net.

Definition 2 (Marking). *The* marking *of a Petri net with identifiers* $N = (P, T, F, H, \alpha, \beta, \gamma)$ *is an element* $M \in \prod_{p \in P} \mathbb{B}(\mathcal{I}^{\alpha(p)})$. *The pair* (N, M) *is called a* marked Petri net. *We define the function* $\mathrm{Id}\colon \prod_{p \in P} \mathbb{B}(\mathcal{I}^{\alpha(p)}) \to \mathcal{P}(\mathcal{I})$ *for markings which returns all identifiers used in marking* M*, i.e.*
$\mathrm{Id}(M) = \{x \in \mathcal{I} \mid \exists p \in P, v \in \mathcal{I}^{\alpha(p)} : x \in rng(v) \wedge M(p)(v) > 0\}$.

A *binding* for a transition t in a marking M gives for each variable an identifier to use. If a variable x is a creator variable for a transition t, i.e. $x \in \mathrm{new}(t)$, the identifier bound by the binding has to be a new identifier, and in the same binding, this identifier cannot be bound to other variables than x.

Definition 3 (Binding). *Let* $N = (P, T, F, H, \alpha, \beta, \gamma)$ *be a Petri net, and* M *a marking. For a transition* $t \in T$*, a* binding $\sigma_{t,M}$ *is a function* $\sigma_{t,M}\colon \Sigma \to \mathcal{I}$*, such that for* $x \in \mathrm{new}(t)$ $\sigma_{t,M}(x) \notin \mathrm{Id}(M)$ *and for* $y \in \mathrm{var}(t)$*, if* $\sigma_{t,M}(x) = \sigma_{t,M}(y)$*, then* $x = y$.

A transition t is enabled, if in a marking M according to a binding $\sigma_{t,M}$ if M can provide tokens described by the binding for "classical" arcs, and it does not contain tokens specified by the binding for inhibitor arcs. If it fires, it produces identifiers according to the arc inscriptions on the outgoing arcs.

Definition 4 (Firing rule for transitions). *A transition* $t \in T$ *is* enabled *in a marking* M *with binding* $\sigma_{t,M}\colon \Sigma \to \mathcal{I}$ *if there is a* $x \in \prod_{p \in {}^\bullet t} \mathcal{I}^{\alpha(p)}$*, such that* $\sigma_{t,M}(\beta(p,t)) = x(p)$*, for all* $p \in {}^\bullet t$*,* $x \leq M$ *and for any* $p \in H(t)$*, there is no binding* $\sigma'_{t,M}\colon \Sigma \to \mathcal{I}$ *with* $\sigma'_{t,M}(x) = \sigma_{t,M}(x)$ *for* $x \in \mathrm{var}(t)$*, such that* $\sigma'_{t,M}(\gamma(p,t)) \in M(p)$*. An enabled transition can* fire *in a marking* M *with binding* σ_t*, resulting in a marking* M'*, denoted as* $M\,[t, \sigma\rangle\, M'$*, if and only if there are* $q \in \prod_{p \in {}^\bullet t} \mathcal{I}^{\alpha(p)}$*, and* $r \in \prod_{p \in t^\bullet} \mathcal{I}^{\alpha(p)}$*, such that* $M' + q = M + r$ *holds and* $\sigma(\beta(p,t)) = q_p$ *for* $p \in {}^\bullet t$*, and* $\sigma(\beta(t,p)) = r_p$ *for* $p \in t^\bullet$.

Let $\mathcal{I} = I\!N$. Transition $t1$ in Figure 1 is enabled e.g. with binding σ_{t1}, where $\sigma_{t1}(x) = 10$, and if $t1$ fires with binding σ_{t1}, the new marking has a token with value 10 in place $p2$. In this marking, transition $t5$ is not enabled with binding σ_{t5} where $\sigma_{t5}(x) = 1$ and $\sigma_{t5}(y) = 5$. However, it is enabled with binding σ'_{t5} where $\sigma_{t5}(x) = 10$ and $\sigma_{t5}(y) = n$, for any $n \in \mathcal{I}$.

In the remainder, we write Petri net for an inhibitor Petri net with identifiers.

4 Entity-Relationship Diagrams

A database consists of entities, elements or records, stored in tables. Between these entities, associations exist. Entities belong to an *entity type*, associations belong to a *relation* between entity types. An Entity-Relationship diagram (ERD) [4], describes the type of the entities and the relations between them. Without loss of generality, we only consider binary relations. The cardinality between a relation r and an entity type E defines the number of associations of type r an entity from E can have, and it is mostly specified as a range. In this paper we limit the cardinality to the set of ranges $C = \{[0..*), [1..*), [0..1], [1..1]\}$. In the remainder, the brackets for the cardinality are omitted.

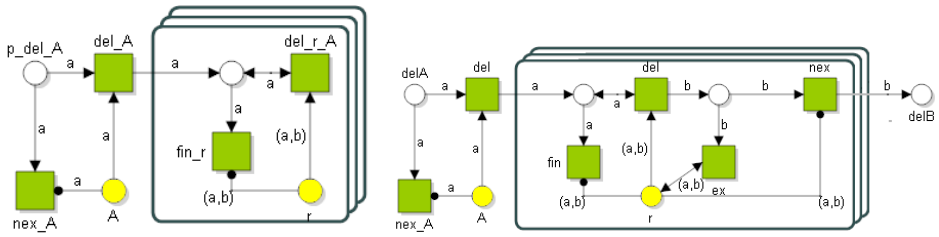
Definition 5 (Entity-Relationship Diagram). *An* Entity-Relationship Diagram *(ERD) S is a 6-tuple $S = (\mathcal{E}, R, \sigma, \tau, C_\sigma, C_\tau)$, where (1) $\mathcal{E}$ and R are two disjoint sets of* entity types *and* relations *respectively; (2) $\sigma, \tau \colon R \to \mathcal{E}$ are the* source *and* target function*, defining the source and target entities of a relation; and (3) $C_\sigma, C_\tau \colon R \to C$ are the* source *and* target cardinality *returning the cardinality of the source and target entity types of the relation.*

The current state of a database is called an instance. An instance consists of entities belonging to an entity type, and relations between these entities.

Definition 6 (Instance of an ERD). *Let $S = (\mathcal{E}, R, \sigma, \tau, C_\sigma, C_\tau)$ be an ERD. An instance I_S of ERD S is a 2-tuple $I_S = (I_E, I_R)$ where (1) $I_E \colon \mathcal{E} \to \mathcal{P}(\mathcal{I})$, returns for each entity type the entities present. An* entity *is identified by an identifier; and (2) $I_R \colon R \to \mathcal{P}(\mathcal{I} \times \mathcal{I})$ returns for each relation the set of associations.*

An instance is consistent if it satisfies the ERD. An instance is said to satisfy the ERD if all the constraints in the ERD are fulfilled, i.e., the cardinalities for each relation are satisfied.

Definition 7 (Consistency of an instance). *Let $S = (\mathcal{E}, R, \sigma, \tau, C_\sigma, C_\tau)$ be an ERD, and $I_S = (I_E, I_R)$ be an instance of this ERD. The instance is* consistent *if for each relation $r \in R$ holds $\pi_1(s) \in I_E(\sigma(r))$, and $\pi_2(s) \in I_E(\tau(r))$. For each entity $A \in \mathcal{E}$ and relation $r \in R$ with $\sigma(r) = A$, holds for all $a \in I_E(A)$ that $|\{b|(a,b) \in I_R(r)\}| \in C_\sigma(r)$. For each entity $B \in \mathcal{E}$ and relation $r \in R$ with $\tau(r) = B$, holds for all $b \in I_E(B)$ that $|\{a|(a,b) \in I_R(r)\}| \in C_\tau(r)$.*



(a) $C_\tau(r) \in \{0..1, 0..*\}$ (b) $C_\tau(r) \in \{1..1, 1..*\}$

Fig. 2. Delete transaction for entity A

On entity types and relations we define the following operations: the insertion of a new entity (associations), the deletion of an entity (association) and the retrieval of an entity (association). With these basic operations, we can create procedures on the instance, possibly resulting in a new instance.

5 Generation of Database Transactions

A transaction is a process using the basic database operations on an instance, such that if the transaction starts with a consistent instance, the resulting instance after the transaction is consistent again. Note that during a transaction the instance need not to be consistent. Transactions should fulfill the ACID properties [12]: *atomicity*, i.e. it should be executed completely or not at all, *consistency*, i.e. it should preserve all constraints, *isolation*, i.e. it is not influenced by other concurrently executed transactions, and *durability*, i.e. changes made should persist.

Transactions have as input a message, and they operate on an instance of the database. A transaction has as output a new instance of the database. In this section, we show how we can generate from an ERD a Petri net that represents a transaction. We consider a transaction in isolation, i.e. management of concurrent transactions is not part of the generated transactions.

Definition 8 (Transaction). *Let $S = (\mathcal{E}, R, \sigma, \tau, C_\sigma, C_\tau)$ be an ERD, and $I_S = (I_E, I_R)$ be an instance of it. A* transaction *on I_S is a a marked Petri net (N, M) with $N = (P_{in} \cup P_{db} \cup P_h, T, F, H, \alpha, \beta, \gamma)$, where P_{in} is the set of* input places*, $P_{db} = (\mathcal{E} \cup R)$ is the set of* database places*, P_h is the set of* internal places*, and P_{in}, P_{db} and P_h are pairwise disjoint. For all places $p \in P_h$ holds $M(p) = \emptyset$, for all places $A \in \mathcal{E} \cap P_{db}$ holds $M(A) = I_E(A)$, and for all places $r \in R \cap P_{in}$ holds $M(r) = I_R(r)$.*

Note that many transactions share the Petri net specification, and only differ in the initial marking.

After the execution of the transaction on a consistent instance, the new instance should be consistent again, and there should not be any garbage in

the net and no transitions of the transaction should be enabled, i.e. in the final marking, there are no transitions enabled, and all internal and input places are empty. We call this property *soundness*.

Definition 9 (Sound transaction). *A transaction is called* sound *if and only if it satisfies the following two properties: (1) for all reachable markings, a final marking is reachable in which all internal and message places are empty; and (2)the final database instance satisfies all cardinality constraints, if the initial instance also satisfied the cardinality constraints.*

Note that the first part of the soundness definition, is in fact the proper completion property of workflow nets [1].

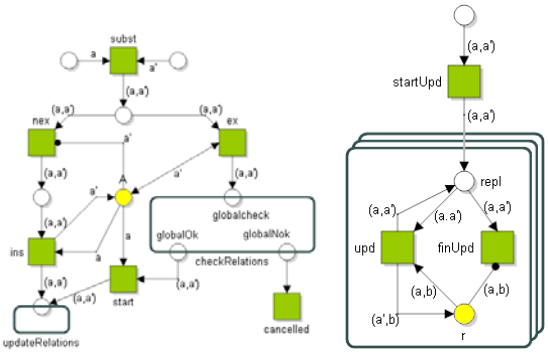
In the remainder of this section, we present algorithms to generate, given an ERD, three types of transactions: (1) delete transactions (subsection 5.1), (2) update transactions (subsection 5.2), and (3) complex insert transactions (subsection 5.3). In all cases, these transactions will be sound, which can easily be verified by inspection of the constructed Petri nets.

5.1 Deletion Transactions

The algorithm generates a Petri net using the construction shown in Figure 2. For each relationship r, there is one subnet (rectangle). The processes in these subnets can be executed concurrently, denoted by three rectangles; nodes outside the rectangles, connected to a node within the rectangles, are connected to each subnet. Multiple entities are deleted, if required by the cardinality constraints. Figure 2 depicts two results of the algorithm, and deletes an entity a of type $A \in \mathcal{E}$, and relation r, with A the source entity of r. First, transition *delA* deletes the entity from A. Next, all associations with a are removed from r by transition *del*, until there are no associations with a left, and transition *fin* is enabled.

Depending on the cardinality of relation r on target entity type B, the associated elements of a need to be deleted. If the relation r is optional for B, the constraints for the associated elements are not violated, and therefore need no check whether they should be removed, as depicted in Figure 2(a). If r is mandatory, transition *del* produces a token with the associated entity b. If b is connected to a different entity of type A, b does not violate the cardinality constraints, which is tested by transition *ex*. If there is no such entity, the associated entity b needs to be deleted, i.e. *nex* is enabled and produces a token with identifier b in place *delB*, indicating entity b needs to be deleted.

The generated transaction is sound, since transition *del* can only fire as many times as there are associations with entity a, which is a bounded number of times. Worst case, the procedure can be called for each associated entity, which is again a bounded number of times. Therefore, eventually, the transaction will reach a marking in which no transition is enabled, and all places, except the places representing the entity types and relations, are empty. The second property of soundness is trivial.



(a) Substitution of an entity (b) Update relations

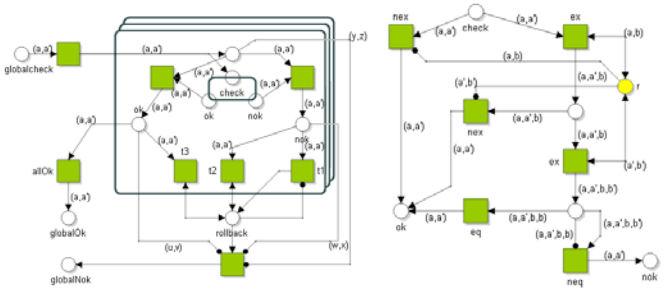
Fig. 3. Outcome of substitution algorithm

5.2 Update Transactions

If an entity is substituted for another entity, all relations the entity is associated with, either as source or as target, needs to be updated as well. The update transaction generated uses the construction shown in Figure 3(a). The transaction takes entities a and a' of type A, where a is the entity to be substituted, and a' the entity to substitute with. If a' does not exist in the current instance, i.e. there is no token a' in place A, enabling transition *nex*, the substitution is no problem, a is substituted by a' by transition *ins*, and all associations with a are transformed into associations with a'. If there is a token a' in place A, transition *ex* is enabled, and each of the relations need to be checked whether the substitution is allowed. The subnet depicted by the rectangle *checkRelations* is replaced by the process depicted in Figure 4(a). If there is a token in place *globalOk*, substitution is allowed by all relations, the token a is removed from place A, and all associations with a are updated. Otherwise, there is a token in place *globalNok*, and transition *cancelled* is enabled.

Figure 3(b) substitutes entity a by entity a' for each association with source a. The rectangle shows the subnet which is repeated for each relation. While there exists an association with a, transition *upd* updates the association and replaces a by a'. If there are no associations with a anymore, transition *finUpd* fires. The pattern for associations with a as target entity type is symmetrical with this pattern: the vector (a, b) is replaced by (b, a) and (a', b) by (b, a').

For each of the relations, the substitution needs to be checked, as depicted in Figure 4(a). The outer rectangle shows the pattern which is generated for each of the relations to check for, and is concurrently added for each relation connected to A. In the rectangle, each relation is checked. If the substitution is allowed, the check produces a token in place *ok*, otherwise a token is produced in place *nok*. If all checks produced a place in their *ok* place, transition *allOk* is enabled, and produces a token in place *globalOk*. If one of the checks produces a token in its *nok* place, an undo, i.e. a rollback, is needed. If there is no token in



(a) Check relations (b) Condition 1

Fig. 4. Check relations whether it can be inserted. Figure (b) is one of the subnets that can be inserted in the checkRelation box of (a)

Table 1. Conditions when a substitution of a by $a' \neq a$ is allowed for relation r

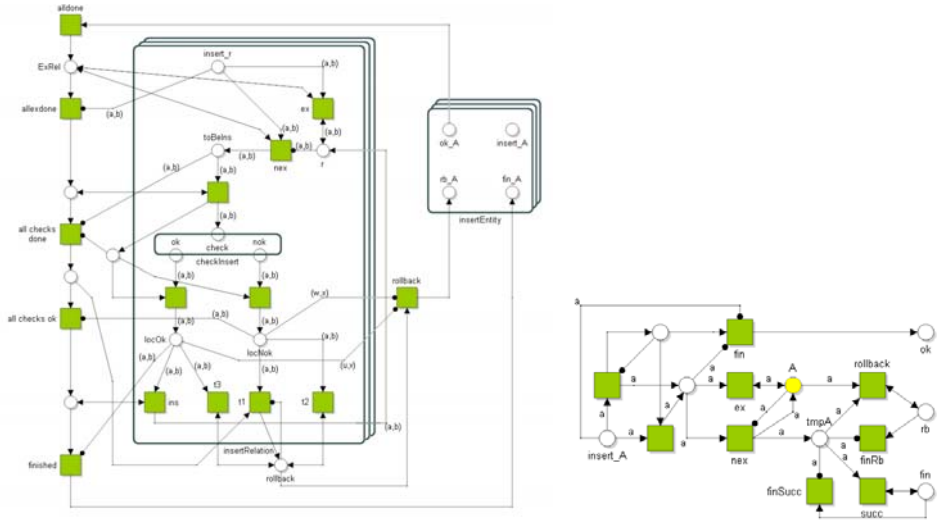
		B			
		0..∗	1..∗	0..1	1..1
A	0..∗	always	always	always	always
	1..∗	always	always	always	always
	0..1	condition 1	condition 1	condition 2	condition 2
	1..1	condition 1	condition 1	never	never

Condition 1: $\forall b, b' \in I_E(B)\colon \{(a,b),(a',b')\} \subseteq I_R(r) \Rightarrow b = b'$
Condition 2: $\forall b, b' \in I_E(B)\colon \neg(\{(a,b),(a',b')\} \subseteq I_R(r))$

place *rollback*, transition $t1$ is enabled, and puts a token in *rollback*. Otherwise, transition $t2$ removes the token from the *nok* place, and transition $t3$ removes the token from the *ok* place. If there are no tokens left in any *ok* or *nok* place, transition *cancelled* is enabled, and produces a token in the place *globalNok*.

Substitution is not allowed if it violates a cardinality constraint. The circumstances under which a constraint is violated, depends on the source and target cardinality of the relation. Let r be a relation from entity type A to entity type B, and let a and a' be two entities of type A present in the instance. Table 1 shows the conditions to allow substitution. If the source cardinality of r is 0..∗ or 1..∗, substitution is always possible. If the cardinality allows for at most one entity of type B, this is not always the case. Suppose $(a,b),(a',b') \in I_R(r)$ are two associations of r. If $b = b'$, substitution is no problem. If $b \neq b'$, a' will be connected with two entities of type B, which is a violation. In case the target cardinality of r also allows for at most 1 entity association, it is never allowed.

Figure 4(b) depicts the check procedure for the first condition in a transaction. If an association (a,b) does not exist, transition *nex* produces a token in place *ok*. If the association exists, transition *ex* fires, and a check is performed whether an association (a',b') exists. If this is not the case, transition *nex* fires and puts a token in place *ok*. Otherwise, transition *ex* fires, and produces a to-



(a) insert transaction (b) Insertion of an entity

Fig. 5. Complex insert transaction. The set of input places are prefixed *insert_*.

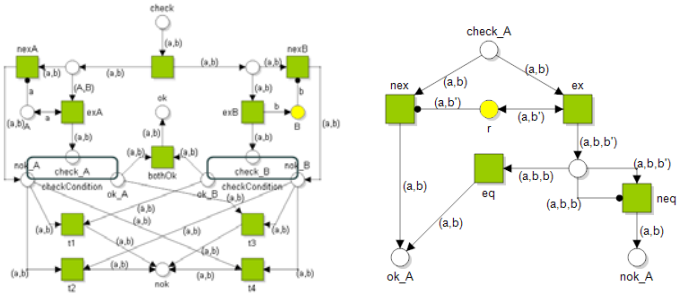
ken with identifiers a, a', b, b'. If b and b' are equal, transition *eq* is enabled, and produces a token in place *ok*. Otherwise, if b and b' are not equal, transition *neq* is enabled, producing a token in place *nok*. In the second condition, where the target cardinality allows for at most one association, the check whether the two entities of type B are equal is omitted.

5.3 Complex Insert Transactions

Inserting a single entity or association is often not possible due to cardinality constraints. Therefore, more complex inserts are needed to make the instance consistent again. The input is a set of entities and associations, according to the ERD. Hence, we define a message to be an (possibly inconsistent) instance of the ERD. The transaction needs to ensure the consistency.

The transaction generated for the insertion of an instance of the ERD is depicted in Figure 5. For each entity type A, there is a place *insert_A*, and for each relation r, there is a place *insert_r*. These places form the input interface of the transaction. Each node outside a rectangle is connected with the corresponding node in each instance of the subnet represented by the rectangle.

The transaction is split into three sequential sub processes. First, all new entities are inserted (the subnets represented by the rectangle *insertEntity*). If all entities are inserted, transition *alldone* is enabled and fires. Rectangle *insertRelation* is executed concurrently for each of the relations. If no association violates any constraint, all associations are inserted. If a constraint is violated the transaction performs a roll back.



(a) Check relation (b) Condition 1

Fig. 6. Check for insertion of a relation, generated by the insert transaction algorithm

Sub process *insertEntity* is depicted in Figure 5(b). If an entity a does not exist (transition *nex*), a token with a is added to place A, together with a token a in place *tmpA*, to be able to do a rollback. If the entity already exists (transition *ex*), nothing is done. If all entities have been checked and inserted, transition *fin* is enabled. A token in place *rb* indicates a rollback. Transition *Rollback* fires until all new entities are removed, and transition *finRb* becomes enabled. A token in place *fin* indicates the successful run of the transaction, so no rollback is needed, and all tokens for the rollback operation are removed, after which *finSucc* becomes enabled.

Transition *nex* moves all new association to place *toBeIns*. If an association is already present, it is ignored. All new associations are checked by a process substituted for the rectangle *checkInsert*. If the association is allowed, a token is produced in place *locOk*, otherwise a token is produced in place *locNok*. If all associations are allowed, i.e. no tokens in place *locNok*, they are inserted by transition *ins*. Otherwise, there is at least one token in place *locNok*, and a rollback is performed, using a same construct as for the update transaction, and putting a token in the place *rb* of subnet *insertEntity*.

Depending on the cardinality constraints, an association can be allowed. Figure 6(a) depicts the subnet for a relation r with source entity type A and target entity type B. If the source (target) entity does not exist, transition *nexA* (*nexB*) will fire. Otherwise, transition *exA* (*exB*) fires, and the conditions are checked. If the outcome is positive, a token is produced in place *ok_A* (*ok_B*), otherwise there is a token in place *nok_A* (*nok_B*). If both checks are positive, transition *bothOk* will produce a token in place *ok*. Otherwise, transitions *t1* to *t4* remove all tokens, and produce a token in place *nok*.

Checking the cardinality constraints is depicted in Figure 6(b). If the cardinalities allow for multiple associations for the entity, adding the association is always allowed. Otherwise, if at most one association is allowed, the conditions as shown in Table 1 hold.

6 Conclusion and Future Work

In this paper, we have presented algorithms to generate insert, delete and update transactions using inhibitor Petri nets with identifiers. This subclass of coloured Petri nets is powerful enough to model complex systems, while analysis and verification of this subclass still remain possible.

The generated transactions automatically preserve the cardinality constraints. These algorithms allow the database designer to generate transactions, taking work off the programmer's hands.

Future work will be to compare this approach with other approaches to generate database transactions, to see to what extend it is possible to verify and analyse transactions. Further research will be to extend the analysis such that it is possible to check for a generated transaction whether the resulting instance is a unique and maximal consistent solution, and to implement a prototype of a relational database manager that generates and the different transactions, and uses them to operate on the database. Up-to-now, we considered cardinality constraints. Next steps will be to also allow cardinalities of the form $[n..m]$ and to consider classes of global constraints, e.g. cyclic constraints, and develop algorithms that generate transactions preserving these constraints.

References

1. W.M.P. van der Aalst. Verification of workflow nets. In *ICATPN 1997*, pages 407–426, London, UK, 1997. Springer-Verlag.
2. A.V. Aho, C. Beeri, and J.D. Ullman. The Theory of Joins in Relational Databases. *ACM Trans. Database Syst.*, 4(3):297–314, 1979.
3. W.W. Armstrong. Dependency Structures of Data Base Relationships. In *IFIP Congress*, pages 580–583, 1974.
4. P.P. Chen. The Entity-Relationship Model: Towards a unified view of Data. *ACM Transactions on Database Systems*, 1:9–36, Jan 1976.
5. E. F. Codd. Further normalization of the data base relational model. *IBM Research Report, San Jose, California*, RJ909, 1971.
6. E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.
7. J. Gray and A. Reuter. *Transaction Processing, concepts and techniques.* Morgan Kaufmann, 1993.
8. Kurt Jensen. *Coloured Petri nets: basic concepts, analysis methods and practical use, vol. 2.* Springer-Verlag, London, UK, 1995.
9. W. Reisig. *Petri Nets: An Introduction*, volume 4 of *Monographs in Theoretical Computer Science: An EATCS Series.* Springer-Verlag, Berlin, 1985.
10. F. Rosa-Velardo, D. de Frutos-Escrig, and O. Marroquín-Alonso. On the expressiveness of mobile synchronizing petri nets. In *SECCO'05*, volume 180 of *ENTCS*, pages 77–94. Elsevier, 2007.
11. F. Rosa-Velardo, O. Marroquín-Alonso, and D. de Frutos-Escrig. Mobile synchronizing petri nets: a choreographic approach for coordination in ubiquitous systems. In *MTCOORD'05*, volume 150 of *ENTCS*, pages 103–126. Elsevier, 2006.
12. A. Silberschatz, H. Korth, and S. Sudarshan. *Database System Concepts.* McGraw Hill, 5th edition, 2006.