Generating Relational Database Transactions From Recursive Functions Defined on EB^3 Traces

Frédéric Gervais
CEDRIC, CNAM-IIE, France
GRIL, Département d'Informatique
Université de Sherbrooke, Canada
frederic.gervais@usherbrooke.ca

Marc Frappier
GRIL, Département d'Informatique
Université de Sherbrooke, Canada
marc.frappier@usherbrooke.ca

Régine Laleau
LACL, Université Paris 12
Département Informatique
IUT Fontainebleau, France
laleau@univ-paris12.fr

Abstract

*EB^3 is a trace-based formal language created for the specification of information systems (IS). Attributes, linked to entities and associations of an IS, are computed in EB^3 by recursive functions on the valid traces of the system. We aim at synthesizing relational database transactions that correspond to EB^3 attribute definitions. Each EB^3 action is translated into a transaction. EB^3 attribute definitions are analysed to determine the key values affected by each action. Some key values are retrieved from **SELECT** statements that correspond to first-order predicates in EB^3 attribute definitions. To avoid problems with the sequencing of SQL statements in the transactions, temporary variables and/or tables are introduced for these key values. Generation of **DELETE** statements is straightforward, but distinguishing updates from insertions of tuples requires more analysis.*

1. Introduction

We are mainly interested in the formal specification of information systems (IS) [12]. In our viewpoint, an IS is a software system that helps an organization to collect and to manipulate all its relevant data. An IS also includes software applications and tools to query and modify the database, to friendly communicate query results to users and to allow administrators to control and modify the whole system. The use of formal methods to design IS [10, 20, 22] is justified by the relevant value of informations and data from corporations like banks, insurance companies, high-tech industries or government organizations.

Currently, the most widely used paradigm for specifying IS is the state transition paradigm. In state-based specifications, a system is generally described by defining state invariant properties that must be preserved by the execution of operations. For instance, Z [3] and B [1] are two examples of state-based formal languages. Existing approaches using state transition for specifying IS include RoZ [4], OMT-B [21] and UML-B [18]. One difficulty with the state transition paradigm is the validation of dynamic properties. Let a, b, c be three events. An event ordering property, like "a, followed by an arbitrary number of b, followed by c", is easier to specify and verify in an event-based model than in a state-based one.

The EB^3 language [10] has been defined for the purpose of specifying IS. EB^3 is an event-based formal language that includes some state-oriented constructs. The EB^3 method provides a different way of specifying IS, which is orthogonal in specification style with respect to state-based formal languages [9]. Moreover, dynamic properties can be easily specified in EB^3 [7], and we plan to implement in the future tools that support the verification of such properties, like in other event-based languages such as CSP [15].

With state-based specifications, refinement techniques are generally used to generate relational implementations, like in [5] for Z and [20] for B specifications. In EB^3, IS are synthesized by interpreting EB^3 process expressions. There already exists an interpreter, called EB3PAI [8], for EB^3 process expressions. However, the computation of attribute values is not taken into account yet. In this paper, we focus

on the synthesis of relational database transactions that correspond to EB^3 attribute definitions. Thus, we will be able to efficiently interpret EB^3 specifications for the purpose of software prototyping and requirements validation. The synthesized programs are of a similar algorithmic complexity as those manually generated by a programmer. Hence, they could also be used in concrete implementations of EB^3 specifications.

Section 2 provides an overview of EB^3. Contrary to other event-based languages like CSP [15], EB^3 extensively uses the concept of state to take the IS data model into account. Attributes are defined by means of recursive functions on the valid traces of the system. EB^3 attribute definitions are presented in Sect. 3. To generate SQL statements from EB^3 attribute definitions, one must not only determine the effects of each action on the attributes, but also the key values to be inserted, updated and/or removed. In Sect. 4, we show how to generate SQL statements that correspond to EB^3 attribute definitions. Finally, Sect. 5 concludes the paper with some comments and perspectives.

2. An Overview of EB^3

The core of EB^3 includes a method and a formal notation to describe a complete and precise specification of the input-output behaviour of an IS. An EB^3 specification consists of the following elements:

1. a user requirements class diagram which includes entity types, associations, and their respective actions and attributes. These diagrams are based on entity-relationship model concepts [6]. In EB^3, the terms *entity type* and *entity* are used instead of class and object.
2. a process expression, denoted by `main`, which defines the valid input traces;
3. recursive functions, defined on the traces of `main`, that assign values to entity and association attributes;
4. input-output rules, which assign an output to each valid input trace.

The denotational semantics of an EB^3 specification is given by a relation R defined on $\mathcal{T}(\texttt{main}) \times O$, where $\mathcal{T}(\texttt{main})$ denotes the traces accepted by `main` and O is the set of output events. Let `trace` denote the system trace, which is a list of valid input events accepted so far in the execution of the system. Let $t::\sigma$ denote the right append of an input event σ to trace t, and let `[]` denote the empty trace. The operational behaviour of `main` is then defined as follows.

`trace` $:=$ `[]`;
forever do
  receive input event σ;
  if `main` can accept `trace::`σ **then**
    `trace` $:=$ `trace::`σ;
    send output event o such that
    $(\texttt{trace}, o) \in R$;
  else
    send error message;

Example. To illustrate the main aspects of this paper, an example of a library management system is introduced. The system has to manage book loans to members. A book is acquired by the library; it can be discarded, but only if it is not lent. A member must join the library in order to borrow a book and he can relinquish library membership only when all his loans are returned or transferred. A member can also transfer a loan to another member. A book can be lent by only one member at once. Figure 1 shows the user requirements class diagram of the example. The signature of EB^3 actions is the following.

```
Acquire(bId:bk_Set,bTitle:T^N):void
Discard(bId:bk_Set):void
Modify(bId:bk_Set,nTitle:T^N):void
DisplayTitle(bId:bk_Set):T^N
Register(mId:mk_Set):void
Unregister(mId:mk_Set):void
Lend(bId:bk_Set,mId:mk_Set):void
Return(bId:bk_Set):void
Transfer(bId:bk_Set,mId:mk_Set):void
```

The special type `void` is used to denote an action with no input-output rule; the output of such an action is always `ok`. Some input parameters can be instantiated by a default value, `NULL`, that denotes undefinedness. The input type is then decorated with "`^N`".

Process Expressions. An input event σ is an instantiation of (the input parameters of) an action `a`. An instantiated action $\texttt{a}(t_1, ..., t_n)$ constitutes an elementary process expression. The special symbol "_" may be used as an actual parameter of an action, to denote an arbitrary value of the corresponding type. Complex EB^3 process expressions can be constructed from elementary process expressions (instantiated actions) using the following operators: sequence (`.`), choice (`|`), Kleene closure (`^*`), interleaving (`|||`), parallel composition (`||`, *i.e.,* CSP's synchronisation on shared actions), guard (`==>`), process call, and quantification of choice (`|x:T:...`) and of interleaving (`|||x:T:...`). The EB^3 notation for process expressions is similar to Hoare's CSP [15]. The complete syntax and semantics of EB^3 can be found in [10].

For instance, the EB^3 process expression for entity type *book* is of the following form:

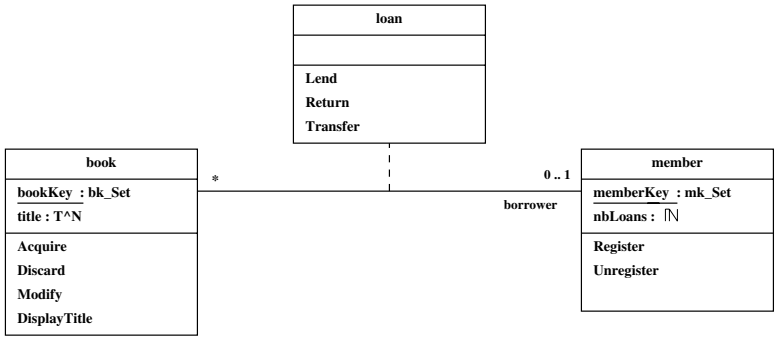

Figure 1. EB^3 specification: User requirements class diagram of the library.

```
book(bId : bk_Set) =
Acquire(bId,_).
(
    ( | mId : mk_Set :
              loan(mId,bId) )^*
 |||
    Modify(bId,_)^*
 |||
    DisplayTitle(bId)^*
).
Discard(bId)
```

where `loan` is the process expression for association $loan$. Firstly, book entity `bId` is produced by action `Acquire`. Then, it can be lent by only one member entity `mId` at once (quantified choice "`| mId : mk_Set : ...`"). Indeed, process expression `book` then calls in turn process expression `loan`, that involves actions `Lend`, `Return` and `Transfer`. The Kleene closure on `loan` means that an arbitrary number of loans can be made on book entity `bId`. At any moment, actions `Modify` and `DisplayTitle` can be interleaved with the actions of `loan`. Action `Modify` is used to change the title of the book, while action `DisplayTitle` outputs the title of the book. Finally, book entity `bId` is consumed by action `Discard`. The complete process expressions for the example are given in [14].

Input-Output Rules. The system trace is usually accessed through recursive functions that extract relevant information from it. Relation R is defined using input-output rules and recursive functions on the system trace. Input-output rules are of the following form:

`RULE` $Name$
`Input` $ActionLabel$
`Output` $RecursiveFunction$
`END;`

For instance, the following input-output rule is defined for action `DisplayTitle`:

```
RULE R
Input DisplayTitle(bId)
Output title(trace,bId)
END;
```

When action `DisplayTitle` is a valid input event, then the recursive function `title` is called to compute the value of attribute $title$. Recursive functions defining attributes are presented in Sect. 3.

3. EB^3 Attribute Definitions

The definition of an attribute in EB^3 is a recursive function on the valid traces, that is, the traces accepted by process expression `main`. This function computes the attribute values. There are two kinds of attributes in a requirements class diagram: key attributes and non-key attributes.

3.1. Defining Key and Non-Key Attributes in EB^3

In the following definitions, we distinguish functional terms from conditional terms. A *functional term* is a term composed of constants, variables and functions of other functional terms. We consider functions using set and arithmetic operators. The data types in which constants and variables are defined are useful basic types like $\mathbb{N}, \mathbb{Z}, \ldots$, Cartesian product of data types and finite powerset of data types. A *conditional term* is of the form **if** $pred$ **then** w_1 **else** w_2 **end**, where $pred$ is a predicate and w_i is either a conditional term or a functional term.

Key Definitions. A key is used in IS to identify entities of entity types or associations: each key value corresponds to a distinct entity of the entity type. Let e be an entity type with a key composed of attributes $k_1, \ldots, k_m$. In EB^3, the key

of e, that is defined by a *single* attribute definition for the set $\{k_1, \ldots, k_m\}$, is a total function $eKey$ of the following form:

$$
\begin{array}{l}
eKey\ (s : \mathcal{T}(\mathsf{main})) : \mathbb{F}(T_1 \times \cdots \times T_m) \ \triangleq \\
\quad \textbf{match}\ last(s)\ \textbf{with} \\
\qquad \begin{array}{lcl}
\bot & : & u_0, \\
a_1(\overrightarrow{p_1}) & : & u_1, \\
\cdots & & \\
a_n(\overrightarrow{p_n}) & : & u_n, \\
\_ & : & eKey(front(s));
\end{array}
\end{array}
$$

where $T_1, \ldots, T_m$ denote the types of $k_1, \ldots, k_m$ and expression $\mathbb{F}(S)$ denotes the set of finite subsets of set S. The recursive function is always given in this CAML-like style (CAML is a functional language [2]). Standard list operators are used, such as *last* and *front* which respectively return the last element and all but the last element of a list; they return the special value $\bot$ when the list is empty.

Expressions $\bot : u_0$, $a_1(\overrightarrow{p_1}) : u_1$, ..., $a_n(\overrightarrow{p_n}) : u_n$, and $\_ : eKey(front(s))$ are called *input clauses*. In an input clause, expression $a_i(\overrightarrow{p_i})$ denotes a *pattern matching* expression, where a_i denotes an action label and $\overrightarrow{p_i}$ denotes a list whose elements are either variables, or the special symbol '_', which stands for a wildcard, or ground functional terms. Special symbol '$\bot$' in input clause $\bot : u_0$ pattern matches with the empty trace, while symbol '_' in $\_ : eKey(front(s))$ is used to pattern match with any list element. Expressions $u_0, \ldots, u_n$ denote functional terms. Let $var(e)$ denote the free variables of e. For each input clause, we should have $var(u_i) \subseteq var(\overrightarrow{p_i})$. The syntax of key definitions is provided by [14].

For example, the key of entity type `book` is defined by:

$$
\begin{array}{l}
bookKey(s : \mathcal{T}(\mathsf{main})) : \mathbb{F}(bk\_Set) \ \triangleq \\
\textbf{match}\ last(s)\ \textbf{with} \\
\quad \bot\ :\ \emptyset, \\
\quad \mathsf{Acquire}(bId)\ :\ bookKey(front(s)) \cup \{bId\}, \\
\quad \mathsf{Discard}(mId)\ :\ bookKey(front(s)) - \{bId\}, \\
\quad \_\ :\ bookKey(front(s));
\end{array}
$$

Non-Key Attributes. A non-key attribute depends on the key of the entity type or of the association. In EB^3, each non-key attribute b_i is defined by a function of the following form:

$$
\begin{array}{l}
b_i\ (s : \mathcal{T}(\mathsf{main}), \overrightarrow{k} : T_1 \times \cdots \times T_m) : T_i \ \triangleq \\
\quad \textbf{match}\ last(s)\ \textbf{with} \\
\qquad \begin{array}{lcl}
\bot & : & u_0, \\
a_1(\overrightarrow{p_1}) & : & u_1, \\
\cdots & & \\
a_n(\overrightarrow{p_n}) & : & u_n, \\
\_ & : & b_i(front(s), \overrightarrow{k});
\end{array}
\end{array}
$$

Parameter $\overrightarrow{k} = (k_1, \ldots, k_m)$ denotes the list of key attributes that define the entity type of b_i, and $T_1, \ldots, T_m$ are the types of $k_1, \ldots, k_m$. The codomain T_i is the type of non-key attribute b_i. It always include $\bot$ to represent undefinedness; hence, EB^3 recursive functions are always total. Moreover, all the functions and operators are strict, *i.e.,* $\bot$ is mapped to $\bot$.

In non-key attribute definitions, expressions $u_0, \ldots, u_n$ denote either functional or conditional terms and, for each input clause, we should have $var(u_j) \subseteq var(\overrightarrow{p_j}) \cup var(\overrightarrow{k})$. Any reference to a key $eKey$ or to an attribute b_j (j can be equal to i) in an input clause is always of the form $eKey(front(s))$ or $b_j(front(s), ...)$. The syntax of non-key attribute definitions can be found in [14].

The next two definitions are two examples of non-key attributes for the library system: $title$ and $nbLoans$.

$$
\begin{array}{ll}
title(s : \mathcal{T}(\mathsf{main}), bId : bk\_Set) : T^{\bot} \ \triangleq & \\
\textbf{match}\ last(s)\ \textbf{with} & \\
\quad \bot\ :\ \bot, & \text{(IC1)} \\
\quad \mathsf{Acquire}(bId, bTitle)\ :\ bTitle, & \text{(IC2)} \\
\quad \mathsf{Discard}(bId)\ :\ \bot, & \text{(IC3)} \\
\quad \mathsf{Modify}(bId, nTitle)\ :\ nTitle, & \text{(IC4)} \\
\quad \_\ :\ title(front(s), bId); & \text{(IC5)}
\end{array}
$$

where $T^{\bot}$ is the mathematical notation for ASCII expression `T^N`.

$$
\begin{array}{l}
nbLoans(s : \mathcal{T}(\mathsf{main}), mId : mk\_Set) : \mathbb{N} \ \triangleq \\
\textbf{match}\ last(s)\ \textbf{with} \\
\quad \bot\ :\ \bot, \\
\quad \mathsf{Register}(mId)\ :\ 0, \\
\quad \mathsf{Lend}(\_, mId)\ :\ 1\ +\ nbLoans(front(s), mId), \\
\quad \mathsf{Return}(bId)\ :\ \textbf{if}\ mId = borrower(front(s), bId) \\
\qquad \textbf{then}\ nbLoans(front(s), mId) - 1 \\
\qquad \textbf{end}, \\
\quad \mathsf{Transfer}(bId, mId')\ :\ \textbf{if}\ mId = mId' \\
\qquad \textbf{then}\ nbLoans(front(s), mId) + 1 \\
\qquad \textbf{else if}\ mId = borrower(front(s), bId) \\
\qquad\quad \textbf{then}\ nbLoans(front(s), mId) - 1\ \textbf{end} \\
\qquad \textbf{end}, \\
\quad \mathsf{Unregister}(mId)\ :\ \bot, \\
\quad \_\ :\ nbLoans(front(s), mId);
\end{array}
$$

3.2. Computation of Attribute Values

When the function associated to attribute b is executed with valid trace s as input parameter, then all the input clauses of the attribute definition are analysed. Let $b(s, v_1, ..., v_n)$ be the attribute to evaluate and ρ be the substitution $\overrightarrow{k} := v_1, ..., v_n$. Each input clause $a_i(\overrightarrow{p_i}) : u_i$ generates a pattern condition of the form

$$
\exists\, (var(\overrightarrow{p_i}) - elem(\overrightarrow{k}))\ \bullet\ last(s) = a_i(\overrightarrow{p_i})\ \rho \qquad (1)
$$

where the right-hand side of the equation denotes the application of substitution ρ on input clause $a_i(\overrightarrow{p_i})$. Expression $elem(\overrightarrow{k})$ denotes the set of elements of $\overrightarrow{k}$. Such a pattern condition holds if the parameters of the last action of trace s match the values of variables $\overrightarrow{k}$ in $\overrightarrow{p_i}$. The first pattern condition that holds in the attribute definition is the one executed. Hence, the ordering of these input clauses is important.

When a pattern condition $a(\overrightarrow{p}) : u$ evaluates to true, an assignment of a value for each variable in $var(\overrightarrow{p})$ has been determined. Functional terms are directly used to compute the attribute value. Predicates of conditional terms determine the last free variables of u in function of the key values and/or the values of $last(s)$. For instance, we have the following values for attribute $title$:

$$title([\,], b_1) \stackrel{(IC1)}{=} \bot$$
$$title([\mathsf{Register}(m_1)], b_1) \stackrel{(IC5)}{=} title([\,], b_1) \stackrel{(IC1)}{=} \bot$$
$$title([\mathsf{Register}(m_1), \mathsf{Acquire}(b_1, t_1)], b_1) \stackrel{(IC2)}{=} t_1$$
$$title([\mathsf{Register}(m_1), \mathsf{Acquire}(b_1, t_1), \mathsf{Modify}(b_1, t_2)], b_1) \stackrel{(IC3)}{=} t_2$$

In the first example, the value is obtained from input clause (IC1), since $last([\,]) = \bot$. In the second example, we first applied the wild card clause (IC5), since no input clause matches Register, and then (IC1). In the third example, the value is obtained immediately from (IC2). In the last example, the value is obtained from (IC3). Since the size of a valid trace is finite and decreases at each recursive call and since the input clause for an empty trace is always defined, then the computation of attribute values terminates.

4. Generating Relational Database Transactions

In the EB^3 semantics, when a new event of action a is accepted by process expression `main`, then all the attributes affected by a must be updated. A relational database transaction is generated for each EB^3 action a. Several analyses are required.

We must firstly analyse the input clauses of EB^3 attribute definitions to determine which attributes are affected by the execution of action a and what the effects of a on these attributes are. In particular, we have to determine the key values to delete from the tables and the key values to update and/or to insert. With the sole input clauses analysis, one cannot distinguish the key values to insert from those to update; an analysis of EB^3 process expressions would be required for that (see Sect. 4.5).

To define the transaction corresponding to a, we must then generate for each table affected by a the SQL statements that correspond to the effects of a, that is the **DELETE**, **UPDATE** and **INSERT** statements, that we call DUI statements. Moreover, we need sometimes to define temporary variables and tables to avoid inconsistencies because of the ordering of the different DUI statements.

The general algorithm is:

```
(1)   create the tables of the database
(2)   initialize the database
(3)   for each action a of the EB3 specification
(4)       analyse the input clauses for a
(5)       define a transaction for a
(6)           generate the SQL definition of all the
(7)           temporary variables and tables
(8)           for each table t affected by a
(9)               generate the DUI statements
(10)          generate a commit
```

Steps (1) and (2) are detailed in Sect. 4.1 and 4.2, respectively. The analysis of the input clauses (line (4)) is presented in Sect. 4.3. The definition of the temporary variables and tables (line (6)) is derived from the analysis of the input clauses (see Sect. 4.3). The definition of transactions (lines (5)-(10)) is discussed in Sect. 4.4.

4.1. Creation of Tables

We use standard algorithms from [6] to create relational tables from the user requirements class diagram. The signature of actions provides the attributes that can be set to **NULL**. If the type of an input parameter of an EB^3 action is decorated with `^N` and if it appears in the class diagram, then the corresponding attribute accepts default value **NULL**. For instance, the table definitions for entity types *book* and *member* are:

```
CREATE TABLE book (
    bookKey bk_Set PRIMARY KEY,
    title T,
    borrower mk_Set REFERENCES member
);
CREATE TABLE member (
    memberKey mk_Set PRIMARY KEY,
    nbLoans INT NOT NULL
);
```

4.2. Initialization

The database initialization is simply a special case of the analysis of the input clauses. Indeed, for each attribute definition b, there exists an input clause of the form $\bot : u$. It denotes the initial value of the attribute and therefore corresponds to the initialization of the tables. The most common value for u is $\emptyset$ for a key. This means that there is no entry in the database table. The most common value for u for a non-key attribute is $\bot$. This means either that there is no

tuple at the initialization or that the entries are initialized to **NULL** for non-key attribute b.

4.3. Analysis of the Input Clauses

The results of the analysis of the input clauses (line (4) of the general algorithm) are the following:

1. the attributes $Att(a)$ affected by action a,
2. the tables $T(a)$ affected by a,
3. for each table t of $T(a)$,
   - the key values $K_{Delete}(t, a)$ to delete from t,
   - the key values $K_{Change}(t, a)$ to insert and/or to update in t.

Algorithm. To obtain $T(a)$, we determine for each attribute b of $Att(a)$ the table of b, denoted by $table(b)$, thanks to the table definitions generated in Sect. 4.1. Then, $T(a)$ is defined by:

$$T(a) = \cup_{b \in Att(a)} table(b)$$

To obtain $K_{Delete}(t, a)$ and $K_{Change}(t, a)$, we determine for each attribute b of $Att(a)$ the corresponding sets $K_D(b)$ and $K_{IU}(b)$ such that:

$$\begin{aligned} K_{Delete}(t, a) &= \cup_{\{b \in Att(a) \wedge table(b) = t\}} K_D(b) \\ K_{Change}(t, a) &= \cup_{\{b \in Att(a) \wedge table(b) = t\}} K_{IU}(b) \end{aligned}$$

$K_D(b)$ and $K_{IU}(b)$ are determined by analysing the input clauses.

The algorithm for analysing the input clauses is the following:

```
determine Att(a)
for each attribute b of Att(a)
    if b is defined by a conditional term
        generate a decision tree for b
        determine SELECT statements for the
        relevant key values
        determine the temporary variables and the
        temporary tables
    determine table(b), K_IU(b), K_D(b)
compute T(a)
for each t in T(a)
    compute K_Delete(t,a),K_Change(t,a)
```

The different steps are explained in the next paragraphs. Since any reference to a key $eKey$ or to an attribute b in an input clause is always of the form $eKey(front(s))$ or $b(front(s), k_1, ..., k_m)$, expression $front(s)$ is now omitted in the next references to recursive functions in the paper, *e.g.,* $eKey()$ and $b(k_1, ..., k_m)$.

Determination of $Att(a)$. Because of the pattern matching analysis described in Sect. 3.2, an attribute b is affected by action a if there exists at least one input clause of the form $a(\overrightarrow{p}) : u$ in the definition of b. This gives us set $Att(a)$. For instance, action Transfer appears in attribute definitions $borrower$ and $nbLoans$; hence, $Att(\mathsf{Transfer}) = \{borrower, nbLoans\}$.

Determination of $K_D(b)$ and $K_{IU}(b)$. For each attribute b of $Att(a)$, there may be several input clauses $a(\overrightarrow{p_j}) : u_j$ with the same label a. $K_D(b)$ and $K_{IU}(b)$ are initialized to $\emptyset$. Then, all the input clauses with the same label a in the attribute definition of b are analysed in their declaration order.

When a pattern condition evaluates to true, an assignment of a value for each variable in $var(\overrightarrow{p_j})$ has been determined. We denote by θ_{u_j} this mapping. For instance, the input clause for action Transfer in attribute $nbLoans$ is of the form:

$$\mathsf{Transfer}(bId, mId') \ : \ ...$$

The assignment is then $\theta = \{bId = bId^\star, mId' = mId^\star\}$, where $bId^\star$ and $mId^\star$ are the formal parameters of action Transfer.

If expression u_j in the input clause is a functional term, then a key value v has been entirely determined. If u_j is $\perp$ or $\emptyset$, then v is added to set $K_D(b)$; otherwise v is added to set $K_{IU}(b)$. For instance, the functional term for input clause Discard in attribute definition $title$ is $\perp$. The corresponding key value is bId. Since $title$ has only one input clause for Discard, we obtain directly $K_D(title) = \{bId\}$.

Nevertheless, if u_j is a conditional term, then we must analyse the different conditions in the **if** predicates. The crux of this analysis is to determine, when event a is received, what are the key values $\{\overrightarrow{v}\}$ such that:

$$b(\mathtt{trace} :: a, \overrightarrow{v}) \neq b(\mathtt{trace}, \overrightarrow{v})$$

The variables in $\overrightarrow{k} \cap var(\overrightarrow{p_j})$ are determined by the pattern mapping θ_{u_j}. The variables in $\overrightarrow{k} - var(\overrightarrow{p_j})$ are determined by the conditions in the conditional term u_j. For instance, the pattern matching of action Transfer in attribute $nbLoans$ does not determine any value for mId. We must then analyse the **if** predicates in the conditional term to determine mId.

We use a binary tree called *decision tree* to analyse the **if** predicates. The leaves of the decision tree are the functional terms in the inner **then** parts of expression u_j, and the edges are the **if** predicates. Thus, by analysing the decision tree, each functional term $ft_{j,i}$ is associated to a set of key values $KV_{j,i}$. If $ft_{j,i}$ is $\perp$, then $KV_{j,i}$ is merged with $K_D(b)$; otherwise, $KV_{j,i}$ is merged with $K_{IU}(b)$. For the sake of concision, we do not deal with decision trees in this paper; their construction and analysis are detailed in [13]. For instance,

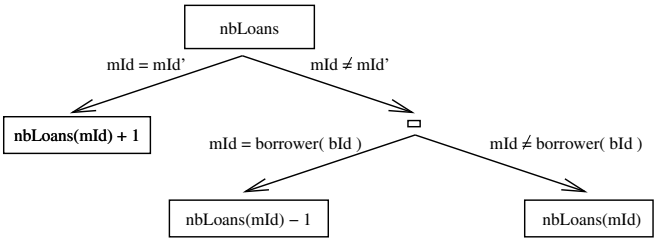

Figure 2. Decision tree of input clause Transfer **in attribute definition** $nbLoans$**.**

the **if** predicates in the conditional term of input clause Transfer in $nbLoans$ determine two key values for mId: mId' and $borrower(bId)$. Figure 2 shows the decision tree obtained for this input clause. The first leaf corresponds to condition $mId = mId'$, and the second leaf to condition $mId \neq mId' \wedge mId = borrower(bId)$. The other leaves are recursive calls of $nbLoans$. The associated functional terms are distinct from $\perp$. In that case, $K_{IU}(nbLoans) = \{mId', borrower(bId)\}$ and $K_D(nbLoans) = \emptyset$.

Definition of Temporary Variables and Temporary Tables. The definition of temporary variables and/or tables is coupled with the analysis of the decision trees. Indeed, when key values are determined from predicates involving arithmetic computations, set computations and/or recursive calls of attributes, then a temporary variable or a temporary table must be defined in the host language, in order to manipulate it in the transaction of the action. Moreover, the temporary variables and tables being defined at the beginning of the transactions, we are free from the DUI statements ordering.

A temporary variable is defined with a **SELECT** statement when a unique key value is determined. For instance, a temporary variable TEMP is introduced as follows[1] for predicate $mId = borrower(bId)$ in the transaction of action Transfer:

```
/* Define a new variable TEMP */
VAR TEMP : mk_Set
    /* Assign the value to TEMP */
    SELECT book.borrower INTO TEMP
    FROM book
    WHERE book.bookKey = #bId;
```

A temporary variable like TEMP can then be simply used in the transaction body by predicates of the form $param =$ TEMP, where $param$ is a parameter in the **WHERE** clauses of SQL statements.

[1]The SQL 92 norm is used for SQL queries, while a procedural pseudo-language is used for transactions.

A temporary table is used to store several key values. For instance, if we need the collection of books lent by member mId, then the following table is defined:

```
CREATE TEMPORARY TABLE
    TAB (bookKey bk_Set PRIMARY KEY);
INSERT INTO TAB
    SELECT book.bookKey
    FROM book
    WHERE book.borrower = #mId;
```

A temporary table like TAB can be used in the transaction body by predicates of the form: $param$ **IN** (**SELECT** TAB.bookKey **FROM** TAB).

SELECT Patterns. The generation of **SELECT** statements that correspond to the key values satisfying the **if** predicates depends on the form of the predicates. We have identified the most typical patterns of predicates and their corresponding **SELECT** statements [14]. In the following definitions, each attribute is prefixed by its table to avoid confusion. Let $table(b)$ denote the table where attribute b is stored and $T.key(j)$ the j-th key attribute of table T.

For a predicate of the form $k = g(\overrightarrow{p})$, where k is a key attribute, g is an attribute recursive call and $\overrightarrow{p}$ is a subset of the input clause parameters, the corresponding **SELECT** statement is:

```
/* extract g */
SELECT table(g).g
FROM table(g)
/* evaluation of g for p_1 */
WHERE table(g).key(1) = #p_1
    AND ...
    /* evaluation of g for p_m */
    AND table(g).key(m) = #p_m;
```

where m is the number of paramaters in $\overrightarrow{p}$. For instance, the **SELECT** statement in temporary variable TEMP above, is an instantiation of this pattern, with predicate $mId = borrower(bId)$.

A more interesting case is a predicate expression of the form $f(\overrightarrow{k}) = g(\overrightarrow{p})$, where f and g are attribute recursive

calls, $\overrightarrow{k}$ is a subset of the key attributes and $\overrightarrow{p}$ is defined as above. The **SELECT** statement for predicate $f(\overrightarrow{k}) = g(\overrightarrow{p})$ is then of the form:

/* extract k_1, ..., k_n */
SELECT $T_f.key(1), ..., T_f.key(n)$
FROM $table(f)\ T_f, table(g)\ T_g$
/* evaluation of g for p_1 */
WHERE $T_g.key(1) = \#p_1$
AND ...
/* evaluation of g for p_m */
AND $T_g.key(m) = \#p_m$
/* predicate */
AND $T_f.f = T_g.g$;

where n is the number of key attributes in $\overrightarrow{k}$. Aliases T_f and T_g are mandatory when f and g are the same attribute.

If g is a composition of attribute recursive calls, then a join between the different tables whose attributes are concerned is necessary. For instance, if $g = g_1; g_2$ (*i.e.,* $f(\overrightarrow{k}) = g_2(g_1(\overrightarrow{p}))$), then a new condition is added in the **SELECT** statement:

SELECT $T_f.key(1), ..., T_f.key(n)$
FROM $table(f)\ T_f, table(g_1)\ T_{g_1}, table(g_2)\ T_{g_2}$
/* new join $g_1; g_2$ */
WHERE $T_{g_2}.key(1) = T_{g_1}.g_1$
AND $T_{g_1}.key(1) = \#p_1$
AND ...
AND $T_{g_1}.key(m) = \#p_m$
AND $T_f.f = T_{g_2}.g_2$;

For instance, let us now suppose that we want to determine the key values mId such that $nbLoans(mId) = nbLoans(borrower(bId))$. Let us apply the pattern above, with $f = g_2 = nbLoans$ and $g_1 = borrower$:

SELECT $M_1.memberKey$
FROM $member\ M_1, book\ B, member\ M_2$
WHERE $M_2.memberKey = B.borrower$
AND $B.bookKey = \#bId$
AND $M_1.nbLoans = M_2.nbLoans$;

4.4. Definition of Transactions

For defining transactions, all the DUI statements are grouped by table. Thanks to the analysis of the input clauses, we have already distinguished the **DELETE** statements from the other statements. The key values to remove from a table t are in set $K_{Delete}(t, a)$. The **DELETE** statements are grouped at the beginning of each table's list of instructions. For each k in $K_{Change}(t, a)$, we determine the list L of **UPDATE** statements for each attribute b of t affected by a, such that $k \in K_{IU}(b)$. We also determine an **INSERT** statement for k and the same set of attributes. Then, we generate the first update in L. A test is defined to determine whether this first **UPDATE** statement has been successful. If so, the other updates are generated. Otherwise, the insertion is executed instead. The subalgorithm for line (9) of the general algorithm is the following:

for each k in $K_{Delete}(t, a)$
determine and generate the **DELETE** statements with k
for each k in $K_{Change}(t, a)$
for each attribute b of t in $Att(a)$
if k is in $K_{IU}(b)$
compute the value of $b(k)$
determine the list L of **UPDATE** statements UPD_l, $1 \leq l \leq p$, for k and the $b(k)$s
determine the **INSERT** statement INS for k and the $b(k)$s
generate UPD_1
generate the following statement :
IF SQL%NotFound
THEN INS
ELSE UPD_2; ... UPD_p;
END;

For instance, the transaction generated for Discard is:

TRANSACTION Discard(bId : bk_Set)
/* delete statement */
DELETE FROM book
WHERE bookKey = #bId;
COMMIT;

This definition is simple, because action Discard only removes key value bId. Nevertheless, when the action involves updates and/or insertions, then the transaction becomes more complex. For instance,

TRANSACTION Acquire(bId : bk_Set,bTitle : T)
/* update statement */
UPDATE book **SET** title = #bTitle
WHERE bookKey = #bId;
/* test to determine whether the update has been successful */
IF SQL%NotFound
THEN
/* insert statement */
INSERT INTO book(bookKey,title)
VALUES (#bId,#bTitle);
END;
COMMIT;

Let us note that the **ELSE** part is not mentionned, because there is only one update in the list generated for Acquire. In practice, action Acquire is a book producer, so the **UPDATE** statement cannot be executed, because entity mId does not exist in the database. In that case, the **IF** predicate is evaluated to true and the **INSERT** statement is executed.

4.5. Optimization

Some transactions can be simplified by analysing the key definitions. Let k be a key value of $K_{Change}(t, a)$. For each non-key attribute b of table t in $Att(a)$, if k is in $K_{IU}(b)$, then we determine in the key definition kd of the key of t whether there exists an input clause for a with symbol $\cup$. If there is no such input clause, then the statement for k is an update, because an insertion requires an union in the key definition. Otherwise, we cannot distinguish insertions from updates without analysing EB^3 process expressions. Indeed, an insertion may be coupled with other updates, or the union specified in the key definition can be redundant.

For instance, by applying the optimization for action Transfer, the results are the following ones. The attributes affected by action Transfer are $borrower$ in table book and $nbLoans$ in table member, $K_{Change}(borrower) = \{bId\}$, and $K_{Change}(nbLoans) = \{mId', borrower(bId)\}$. Neither $memberKey$ nor $bookKey$ includes an input clause for Transfer. Consequently, the SQL statements are updates:

TRANSACTION Transfer(bId:bk_Set,mId:mk_Set)
VAR TEMP : mk_Set
SELECT borrower **INTO** TEMP
FROM book
WHERE bookKey = #bId;
UPDATE book **SET** borrower = #mId
WHERE bookKey = #bId;
UPDATE member **SET** nbLoans = nbLoans + 1
WHERE memberKey = #mId;
UPDATE member **SET** nbLoans = nbLoans − 1
WHERE memberKey = TEMP;
COMMIT;

Thanks to this optimization, we avoid the definition of two **IF** statements and of three insertions in transaction Transfer. Nevertheless, such an analysis is not sufficient to avoid the **IF** statement in transaction Acquire (see Sect. 4.4).

5. Conclusion

5.1. Related Works

Several papers deal with the synthesis of relational implementations. Most of the time, refinement techniques are used to implement specifications using the state transition paradigm [5, 20]. There is some work on the synthesis of systems by interpretation and/or simulation [11, 19], but these tools are generally inefficient for IS. EB3PAI [8] is an interpreter for EB^3 process expressions. It allows one to generate IS from EB^3 specifications. Nevertheless, the computation of attribute values in EB3PAI was not taken into account yet, and transactions were considered as black boxes.

Concerning the synthesis of **SELECT** statements from the predicates of conditional terms, our work is close to Hohenstein's SQL/EER language [17]. SQL/EER is a formal query language that can be automatically translated into SQL [16]. Nevertheless, contrary to SQL/EER, EB^3 provides formal techniques and notations to specify both the functional behaviour and the data model of IS.

5.2. Results and Limits

We have presented an overview of an algorithm that generates relational database transactions from EB^3 attribute definitions. Synthesized transactions can be used in concrete implementations of EB^3 specifications; their algorithmic complexity is similar to those of manually written programs. Our programs introduce some overhead, because they systematically store the current values of attributes before updating the database, in order to ensure correctness. We plan to optimize these programs by analysing dependencies between update statements and avoid, when possible, these intermediate steps.

Let us note that, although the semantics of EB^3 is based on traces, we do not need to keep track of the system trace in the synthesized transactions. The current values of the system trace and of the EB^3 attributes are represented by the current state of the database. Hence, synthesized programs are efficient in terms of space complexity.

The verification of data integrity constraints in EB^3 is an open issue. In particular, static integrity constraints are safety properties and it is difficult to verify safety properties on recursive functions [9]. For instance, a static data integrity constraint for attribute $nbLoans$ is the following: the number of loans of each member is limited to five books. A first option would be the definition of new input-output rules to abort an input event that is valid for EB^3 process expressions, but that violates data integrity constraints. Such an analysis would be made on the fly, during the interpretation. Another option is the definition of guards in EB^3 process expressions. A state-based model would then be required to verify static data integrity constraints. In [13], we use the B language [1] to represent EB^3 attribute definitions and to verify safety properties.

5.3. Perspectives

As a future work, we plan to implement tools that support the generation of relational database transactions from EB^3 attribute definitions. We also plan to combine this translation with the interpretation of EB^3 process expressions in order to optimize the resulting transactions and to synthesize IS that take the computation of attribute values

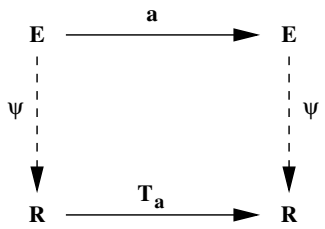

Figure 3. Correctness of transactions.

into account. By interpreting process expressions, we will be able to optimize the transactions obtained in this paper. For instance, the **producer-modifier-consumer** pattern [10] is a usual pattern for entity type process expressions in EB^3. An action like Acquire is considered as a book producer, and consequently, the corresponding transaction can only insert new values into table book. Likewise, a modifier like Modify updates some attribute values, while a consumer like Discard removes some tuples from the table of the entity type.

We aim at validating our translation by proving its correctness. For each EB^3 action a, we know how to generate a transaction T_a. This corresponds to our translation rules. We will model each action a as a transition from E to E, where E is the state space of the IS in the EB^3 model. Analogously, T_a can be considered as a transition from R to R, where R is the state space of the IS in the relational database model. To prove correctness, we want to define a morphism Ψ from the EB^3 model to the relational database model such that the diagram in Fig. 3 is commutative.

References

[1] J.-R. Abrial. *The B-Book: Assigning programs to meanings*. Cambridge University Press, 1996.

[2] G. Cousineau and M. Mauny. *The functional approach to programming*. Cambridge University Press, Cambridge, 1998.

[3] J. Davies and J. Woodcock. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall, 1996.

[4] S. Dupuy, Y. Ledru, and M. Chabre-Peccoud. An overview of RoZ: a tool for integrating UML and Z specifications. In *12th Int. Conf. CAiSE'00*, volume 1789 of *LNCS*, pages 417–430, Stockholm, Sweden, June 2000. Springer-Verlag.

[5] D. Edmond. Refining database systems. In *Proc. ZUM'95*, volume 967 of *LNCS*, Limerick, Ireland, September 1995. Springer-Verlag.

[6] R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 2004.

[7] N. Evans, H. Treharne, R. Laleau, and M. Frappier. How to verify dynamic properties of information systems. In *2nd IEEE International Conference SEFM*, Beijing, China, September 2004. IEEE Computer Society Press.

[8] B. Fraikin and M. Frappier. EB3PAI: an interpreter for the EB3 specification language. In *Proc. 15th International Conference on Software and Systems Engineering and their Applications*, Paris, France, December 2002.

[9] B. Fraikin, M. Frappier, and R. Laleau. State-based versus event-based specifications for information systems: a comparison of EB^3 and B. *Software and System Modeling*, to appear.

[10] M. Frappier and R. St-Denis. EB^3: an entity-based black-box specification method for information systems. *Software and Systems Modeling*, 2(2):134–149, July 2003.

[11] H. Garavel and J. Sifakis. Compilation and verification of LOTOS specifications. In *Proc. 10th International Symposium on Protocol Specification, Testing and Verification*, pages 379–394, Ottawa, Canada, June 1990.

[12] F. Gervais. *EB^4 : Vers une méthode combinée de spécification formelle des systèmes d'information*. Dissertation for the general examination, GRIL, Université de Sherbrooke, Québec, June 2004.

[13] F. Gervais, M. Frappier, and R. Laleau. *Synthesizing B substitutions for EB^3 attribute definitions*. Technical Report 683, CEDRIC, Paris, France, November 2004.

[14] F. Gervais, M. Frappier, R. Laleau, and P. Batanado. *EB^3 attribute definitions: Formal language and application*. Technical Report 700, CEDRIC, Paris, France, February 2005.

[15] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[16] U. Hohenstein. Automatic transformation of an entity-relationship query language into SQL. In *Proc. 8th International Conf. on Entity-Relationship Approach*, Toronto, Canada, 1989. Elsevier.

[17] U. Hohenstein and G. Engels. SQL/EER — Syntax and semantics of an entity-relationship-based query language. *Information Systems*, 17(3):209–242, May 1992.

[18] R. Laleau and A. Mammar. An overview of a method and its support tool for generating B specifications from UML notations. In *ASE: 15th IEEE Conference on Automated Software Engineering*, Grenoble, France, September 2000. IEEE Computer Society Press.

[19] M. Leucker and T. Noll. Rapid prototyping of specification language implementations. In *Proc. 10th IEEE International Workshop on Rapid System Prototyping*, pages 60–65. IEEE Society Press, 1999.

[20] A. Mammar. *Un environnement formel pour le développement d'applications base de données*. PhD thesis, CNAM, Paris, 2002.

[21] E. Meyer and J. Souquières. A systematic approach to transform OMT diagrams to a B specification. In *FM'99*, volume 1708 of *LNCS*, Toulouse, France, September 1999. Springer-Verlag.

[22] H. Nguyen. *Dérivation de spécifications formelles B à partir de spécifications semi-formelles*. PhD thesis, CNAM, 1998.