CEDRIC

Technical Report 700:

EB^3 Attribute Definitions: Formal Language and Application

Frédéric Gervais[1,2]
Marc Frappier[2]
Régine Laleau[3]
Panawé Batanado[2]

First version in February 2005
Version 4 - April 2006

[1] CEDRIC, CNAM-IIE,
18 Allée Jean Rostand, 91025 Évry Cedex, France
gervais@iie.cnam.fr

[2] GRIL, Université de Sherbrooke,
2500, Boulevard de l'Université
Sherbrooke (Québec) J1K 2R1, Canada
{marc.frappier,frederic.gervais}@usherbrooke.ca
panawe@yahoo.fr

[3] LACL, Université Paris 12,
IUT Fontainebleau, Département Informatique
Route Forestière Hurtault, 77300 Fontainebleau, France
laleau@univ-paris12.fr

Abstract

EB^3 is a trace-based formal language created for the specification of information systems (IS). In this technical report, we present the EB^3 formal language for attribute definitions. Attributes, linked to entities and associations of an IS, are computed in EB^3 by recursive functions on the valid traces of the system. The syntax and the main properties of the language are introduced. Then, we aim at synthesizing imperative programs that correspond to EB^3 attribute definitions. Thus, each EB^3 action is translated into a transaction. EB^3 attribute definitions are analysed to determine the tables and the key values affected by each action. Some key values are determined from **SELECT** statements that correspond to first-order predicates in EB^3 attribute definitions. To avoid inconsistencies because of the sequencing of SQL statements in the transactions, temporary variables and/or tables are introduced for these key values. We show the main patterns for the **SELECT** statements used in the temporary variables and/or tables. The SQL statements are then ordered by table. Generation of **DELETE** statements is straightforward, and tests are defined in the transactions to distinguish updates from insertions of tuples. Our algorithms are illustrated by an example of a library management system. Finally, we briefly present the tool called EB^{3}TG, which implements the algorithms introduced in this report.

Contents

Chapter 1

Introduction

We are mainly interested in the formal specification of information systems (IS) and in the synthesis of their implementation. 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 a database (DB), 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 [FSD03, Mam02, Ngu98] is justified by the relevant value of data from corporations like banks, insurance companies, high-tech industries or government organizations. EB^3 [FSD03] is a trace-based formal language created for the specification of IS. The behaviour of the system is specified by means of process expressions that represent the finite traces accepted by the system. Attributes, linked to entities and associations of the IS, are computed in EB^3 by recursive functions on the valid traces of the system.

1.1 An overview of EB^3

An EB^3 specification consists of the following elements:

1. A diagram describes the entity types and associations of the IS, and their respective actions and attributes. It is based on entity-relationship (ER) model concepts [Elm04] and uses a subset of the UML graphical notation for class diagrams. This graphic is called ER diagram in the remainder of the report.

2. A process expression, denoted by *main*, defines the valid input traces of the system.

3. Input-output (I/O) rules assign an output to each valid input trace of the system. Let R denote the set of I/O rules.

4. Recursive functions, defined on the valid input traces of *main*, assign values to entity type and associations attributes.

5. A graphical user interface (GUI) specification describes the functionalities of Web interfaces used to interact with IS end-users.

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 comprised 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 an EB^3 specification is 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 (trace, o) ∈ R;
    else
        send error message;
```

An input event σ is an instantiation of (the input parameters of) an action. The signature of an action `a` is given by a declaration

$$\texttt{a}(q_1 : T_1, \ldots, q_n : T_n) : (q_{n+1} : T_{n+1}, \ldots, q_m : T_m)$$

where $q_1, \ldots, q_n$ are input parameters of types $T_1, \ldots, T_n$ and $q_{n+1}, \ldots, q_m$ are output parameters of types $T_{n+1}, \ldots, T_m$. An instantiated action $\texttt{a}(t_1, ..., t_n)$ also 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 interleaving (`|||x : T : ...`). The EB^3 notation is similar to CSP [Hoa85] but the main differences between EB^3 and CSP are: i) EB^3 allows one to use a single state variable, the system trace, in predicates of guard statements; ii) EB^3 uses a single operator, concatenation (as in regular expressions), instead of prefixing and sequential composition, which makes specifications easier to read and write.

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.

The definition of a recursive function is written in a functional language style (CAML) and traverses the system trace which is represented by a list. 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 `nil` when the list is empty. Finally the symbol "_" is used to pattern match with any list element. The main principles of EB^3 are described in [FSD03]. In particular, the syntax and the semantics of EB^3 process expressions are defined in that paper.

1.2 Outline

Chapter 2 is an introduction to the EB³ formal language for attribute definitions. Then, the syntax of the language is presented in Chapter 3. Chapter 4 shows how to generate relational DB transactions that correspond to EB³ attribute definitions. Then, Chapter 5 presents an overview of the tool EB³TG which implements the algorithms introduced in Chapter 4. Finally, Chapter 6 concludes this report with some perspectives of our work.

To illustrate the main aspects of this report, we introduce an example of a library management system. The library system has to manage book loans to members:

1. A book is acquired by the library. It can be discarded, but only if it is not borrowed.
2. A member must join the library in order to borrow a book.
3. A member can transfer a loan to another member.
4. A member can relinquish library membership only when all his loans are returned or transferred.
5. A member can reserve a book if it is borrowed by another member.
6. A book can be borrowed by only one member at once.
7. A book can be reserved, even by several members, but only if it is borrowed.
8. A book can be taken by the first member that has reserved it, once it has been returned.
9. Loans and transfers are parameterized by the type of loans: permanent or classic.

Figure 1.1 shows the ER diagram used to construct the specification. By studying the requirements, we identify two main entity types, `member` and `book`, and two associations between them, `loan` and `reservation`, with their corresponding actions and attributes. Figure 1.2 provides the signature of actions. 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`.

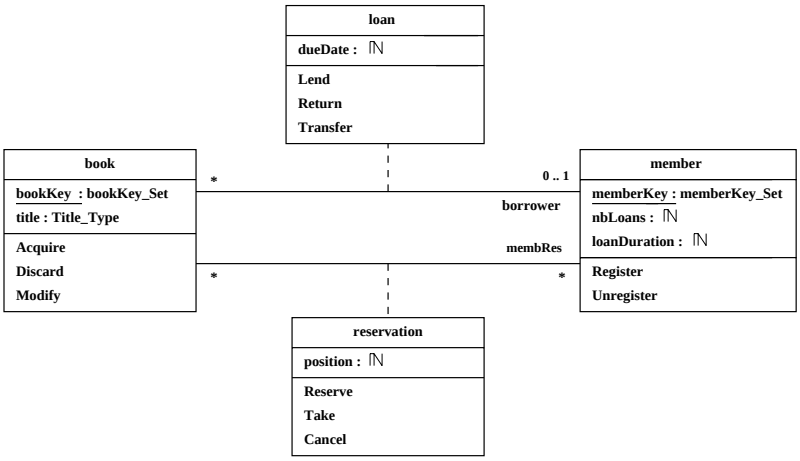

Figure 1.1: ER diagram of the library

```
Acquire(bId:bookKey_Set,bTitle:Title_Type^N):void
Discard(bId:bookKey_Set):void
Modify(bId:bookKey_Set,newTitle:Title_Type^N):void
Register(mId:memberKey_Set,lD:NAT):void
Unregister(mId:memberKey_Set):void
Lend(bId:bookKey_Set,mId:memberKey_Set,typeOfLoan:Loan_Type):void
Return(bId:bookKey_Set):void
Transfer(bId:bookKey_Set,mId:memberKey_Set,typeOfLoan:Loan_Type):void
Reserve(bId:bookKey_Set,mId:memberKey_Set):void
Take(bId:bookKey_Set,mId:memberKey_Set,typeOfLoan:Loan_Type):void
Cancel(bId:bookKey_Set,mId:memberKey_Set):void
```

Figure 1.2: Signature of EB^3 actions

Chapter 2

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 ER diagram: key attributes and non-key attributes.

Conventions

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. The data types in which constants and variables are defined can be abstract or enumerated sets, 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. Hence, a conditional term can include nested **if** statements, whereas a functional term cannot contain an **if** statement.

Expression $var(e)$ denotes the free variables of expression e. A *ground* term t is a term without free variables. Thus, $var(t) = \emptyset$.

2.1 Key Attributes

In EB^3, a key is used to identify instances 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 $k_1, \ldots, k_m$ and non-key attributes $b_1, \ldots, b_n$. In EB^3, the key of entity type e, that is defined by a *single* attribute definition for the set $\{k_1, \ldots, k_m\}$, is a total function $eKey$ of type

$$eKey \ : \ \mathcal{T}(\mathsf{main}) \rightarrow \mathbb{F}(T_1 \times \cdots \times T_m)$$

where $T_1, \ldots, T_m$ denote the types of key attributes $k_1, \ldots, k_m$ and expression $\mathbb{F}(S)$ denotes the set of finite subsets of set S. For instance, the type of the function defining the key of entity type `book` (see Fig. 1.1) is

$$bookKey \ : \ \mathcal{T}(\mathsf{main}) \rightarrow \mathbb{F}(bookKey\_Set)$$

In EB^3, the recursive function defining an attribute is always given in a CAML-like style [CM98]. The definition of the key of entity type e has the following form.

$$\begin{array}{l} eKey\ (s : \mathcal{T}(\mathsf{main})) : \mathbb{F}(T_1 \times \cdots \times T_m) \ \triangleq \\ \quad \textbf{match}\ last(s)\ \textbf{with} \\ \qquad \begin{array}{lcl} \bot & : & u_0, \\ a_1(\overrightarrow{p_1}) & : & u_1, \\ \cdots & & \\ a_n(\overrightarrow{p_n}) & : & u_n, \\ \_ & : & eKey(front(s)); \end{array} \end{array} \tag{2.1}$$

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*. Expression $last(s)$ returns $\bot$ when s is the empty sequence.

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. Expressions $u_0, \ldots, u_n$ denote functional terms. For each input clause, we must have $var(u_i) \subseteq var(\overrightarrow{p_i})$.

For key attribute definitions, a functional term of an input clause $a_i(\overrightarrow{p_i}) : u_i$ is a set expression of the following form:

- $\emptyset$, a constant which denotes there is no entity,
- S, a set which represents the existing entities,
- $eKey(front(s)) \cup S$, which denotes the addition of new entities,
- $eKey(front(s)) - S$, which denotes the removal of entities.

Set S is composed of elements of the following form:

- c, a constant from type $T_1 \times \cdots \times T_m$,
- v, a variable from $var(\overrightarrow{p_i})$.

Let ass denote a l-ary association between l entity types $e_1, \ldots, e_l$ with non-key attributes $b_1, \ldots, b_n$. In general, the key of association ass is formed with the keys of its corresponding entity types. For every $j \in \{1, 2, \ldots, l\}$, let us note $k^j_1, \ldots, k^j_{m_j}$ the key attributes of entity type e_j. The key of ass is then defined by considering all the key attributes of entity types $e_1, \ldots, e_l$:

$$ass \ : \ \mathcal{T}(\mathsf{main}) \rightarrow \mathbb{F}(T_{k^1_1} \times \cdots \times T_{k^1_{m_1}} \times \cdots \times T_{k^l_1} \times \cdots \times T_{k^l_{m_l}})$$

For instance, the key of association `reservation` is of type:

$$reservation \ : \ \mathcal{T}(\mathsf{main}) \rightarrow \mathbb{F}(bookKey\_Set \times memberKey\_Set)$$

When the multiplicity of an association contains 0..1 or 1, then the key of the association is even simpler. The standard rules and algorithms can be found in [Elm04]. For instance, the key of association `loan` is of type:

$$loan \ : \ \mathcal{T}(\mathsf{main}) \rightarrow \mathbb{F}(bookKey\_Set)$$

because entity type `member` has a multiplicity of 0..1 and attribute *borrower* denotes the unique borrower of a book.

2.2 Non-Key Attributes

In EB³, each non-key attribute b_i of an entity type or an association is defined by a function b_i of type

$$b_i \ : \ \mathcal{T}(\mathsf{main}) \times T_1 \times \cdots \times T_m \to T_i$$

where $T_1, \ldots, T_m$ denote the types of all the key attributes of the entity type or the association and the codomain T_i is the type of non-key attribute b_i. The codomain of a non-key recursive function always include $\perp$, to represent undefinedness. Hence, EB³ recursive functions are total. For instance, the type of the function associated to attribute *nbLoans* is

$$nbLoans \ : \ \mathcal{T}(\mathsf{main}) \times memberKey\_Set \to \mathbb{N}$$

The definition of a non-key attribute b_i has the following form.

$$\begin{array}{l} b_i\ (s : \mathcal{T}(\mathsf{main}),\ \overrightarrow{k} : T_1 \times \cdots \times T_m) : T_i \ \triangleq \\ \quad \textbf{match}\ last(s)\ \textbf{with} \\ \qquad \begin{array}{lcl} \perp & : & u_0, \\ a_1(\overrightarrow{p_1}) & : & u_1, \\ \cdots & & \\ a_n(\overrightarrow{p_n}) & : & u_n, \\ \_ & : & b_j(front(s),\ \overrightarrow{k}); \end{array} \end{array} \tag{2.2}$$

where b_j is an attribute (j can be equal to i). Expression $\overrightarrow{k}$ denotes the list of key attributes. Expressions $u_0, \ldots, u_n$ denote either functional or conditional terms. For each input clause, we must have $var(u_j) \subseteq var(\overrightarrow{p_j}) \cup var(\overrightarrow{k})$.

For non-key attribute definitions, a functional term in expression u_j is one of the following forms:

- $\perp$, a constant which denotes an undefined value for the attribute,
- c, a constant from type T_i, which denotes the value of the attribute,
- v, a variable from $var(\overrightarrow{p_j}) \cup var(\overrightarrow{k})$,
- $g(\overrightarrow{t})$, where each $t_i \in \overrightarrow{t}$ is a functional term and g is either an attribute of the ER diagram or an operator on one or more of the different types of the attributes.

The last expression, $g(\overrightarrow{t})$, allows us to define recursive calls to b_i and/or calls to other attribute definitions. A reference to a key $eKey$ or to an attribute b in an input clause is always of the form $eKey(front(s))$ or $b(front(s), ...)$. Moreover, operators must be of relevant types such that the computation of each expression u_j is of type T_i.

2.3 Syntactical Conventions

In practice, attribute definitions are syntactically simplified by omitting expression **match** $last(s)$ **with** and parameter s, that represents a valid trace of the system.

Moreover, the first input clause (*i.e.,* $\perp$) may be also omitted. In that case, a default value is returned for the function: $\perp$ for a non-key attribute and $\emptyset$ for a key attribute. The last input clause (*i.e.,* '_') is never mentioned in the attribute definition. Since '_' is a wildcard, the last clause denotes that the value is always $eKey(front(s))$ for a key or $b(front(s), \overrightarrow{k})$ for a non-key attribute. When an **else** is omitted in an **if** statement, its default values are also $eKey(front(s))$ and $b(front(s), \overrightarrow{k})$, respectively. More generally, any references to $eKey()$ and $b(\overrightarrow{k})$ in the input clauses stand for $eKey(front(s))$ and $b(front(s), \overrightarrow{k})$, respectively.

Since key definitions only differ from non-key attribute definitions by some syntactic restrictions to functional terms and by no input parameter outside valid trace s, we use the following general form for all kinds of attribute definitions in the remainder of the report:

$$\begin{array}{lll} b\,(\overrightarrow{k}) : T \;\triangleq & & \\ \quad \perp & : & u_0, \\ \quad a_1(\overrightarrow{p_1}) & : & u_1, \\ \quad \ldots & & \\ \quad a_n(\overrightarrow{p_n}) & : & u_n; \end{array} \tag{2.3}$$

where $\overrightarrow{k}$ is either $\emptyset$ if b is the definition of a key, or the list of key attributes if b is a non-key attribute, and T is the type of the output values.

2.4 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 $b(s, v_1, \ldots, v_n)$ be the attribute to evaluate and ρ be the substitution $\overrightarrow{k} := v_1, \ldots, v_n$. Each input clause $a_i(\overrightarrow{p_i}) : u_i$ generates a pattern condition of the form

$$\exists\,(var(\overrightarrow{p_i}) - \overrightarrow{k}) \bullet last(s) = a_i(\overrightarrow{p_i})\ \rho \tag{2.4}$$

where the right-hand side of the equation denotes the application of substitution ρ on input clause $a_i(\overrightarrow{p_i})$. 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.

An attribute b may be affected by an input event $\sigma = a(\sigma_1, \ldots, \sigma_m)$ if there exists an input clause $a(\overrightarrow{p}) : u$ in the definition of b. There may be several input clauses with the same label a:

$$\begin{array}{lll} b\,(\overrightarrow{k}) : T \;\triangleq & & \\ \quad \ldots & & \\ \quad a(\overrightarrow{p_1}) & : & u_1, \\ \quad \ldots & & \\ \quad a(\overrightarrow{p_n}) & : & u_n; \\ \quad \ldots & & \end{array} \tag{2.5}$$

Since the first input clause that evaluates to true is the one to be executed, analysis of input clauses is done in their declaration order.

For each input clause of the form $a(\overrightarrow{p}) : u$, the goal is to identify the free variables of $\overrightarrow{p}$ with the actual parameters $\sigma_1, \ldots, \sigma_m$ of σ. The pattern condition (2.4) can be rewritten into a conjunction of equalities. Let p_j denote the j^{th} formal parameter of input clause $a(\overrightarrow{p})$, with $j \geq 1$.

$$\exists\, (var(\overrightarrow{p}) - \overrightarrow{k}) \bullet \bigwedge_j p_j = \sigma_j$$

There are three cases to consider to evaluate this condition:

1. when p_j is the wildcard symbol '_', its corresponding equality trivially holds;
2. when p_j is a ground term, the equality $p_j = \sigma_j$ can be evaluated by simply computing the value of p_j and comparing it to σ_j;
3. when p_j is a variable, we can set the value of p_j to σ_j to satisfy the equality; however, if this variable v occurs in more than one place in $\overrightarrow{p}$, then we must check that the corresponding values in σ are all the same. In other words, let J_v be the list of indices where v occurs in $\overrightarrow{p}$, then:

$$\bigwedge_{j_1, j_2 \in J_v} \sigma_{j_1} = \sigma_{j_2}$$

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. We denote by θ these value assignments; θ is in fact a substitution. Thus, the value of attribute b can be computed with the corresponding expression u, in which the free variables depending on $\overrightarrow{p}$ are assigned the corresponding values in σ, thanks to substitution θ. Functional terms are directly used to compute the attribute value. Predicates of conditional terms determine the last free variables of u from the key values and/or the values of σ.

For example, the definition of attribute *title* is:

$$\begin{array}{l} title(s : \mathcal{T}(\mathsf{main}), bId : bookKey\_Set) : Title\_Type \triangleq \\ \quad \textbf{match } last(s) \textbf{ with} \\ \qquad \begin{array}{lcll} \bot & : & \bot, & \text{(I1)} \\ \mathsf{Acquire}(bId, bTitle) & : & bTitle, & \text{(I2)} \\ \mathsf{Discard}(bId) & : & \bot, & \text{(I3)} \\ \mathsf{Modify}(bId, nTitle) & : & nTitle, & \text{(I4)} \\ \_ & : & title(front(s), bId); & \text{(I5)} \end{array} \end{array}$$

With the default input clauses (*e.g.,* (I1) and (I5)), each attribute function is total, *i.e.,* it is defined for any input trace s and any key value. For instance, we have the following values for *title*.

$$\begin{array}{l} title([\,], b_1) \overset{\text{(I1)}}{=} \bot \\ title([\mathsf{Register}(m_1, 21)], b_1) \overset{\text{(I5)}}{=} title([\,], b_1) \overset{\text{(I1)}}{=} \bot \\ title([\mathsf{Register}(m_1, 21), \mathsf{Acquire}(b_1, t_1)], b_1) \overset{\text{(I2)}}{=} t_1 \end{array}$$

In the first case, the value is obtained from input clause (I1), since $last([\,]) = \bot$. In the second case, we first apply the wild card clause (I5), since no input

Figure 2.1: Roles in a $M : N$ association

Figure 2.2: Roles in a $1 : N$ association

clause matches Register, and then (I1). In the third casee, the value is obtained immediately from (I2) with the following substitution:

$$\theta = \{bId := b_1\}$$

2.5 Properties

Auxiliary Functions

EB^3 process expressions can use auxiliary recursive functions on the trace to determine some (set of) values. An auxiliary function f is similar to an attribute definition, but it can return either scalars, sets of values or lists. It is defined by a recursive function of type

$$f \ : \ \mathcal{T}(\mathsf{main}) \times T_1 \times \cdots \times T_m \rightarrow T$$

where $T_1, \ldots, T_m$ denote the types of some key attributes and the codomain T is the type of the output values of f. Like attribute definitions, an auxiliary function is total and is defined with a CAML-like pattern matching. Their use is not allowed in attribute definitions. They are used in EB^3 process expressions. Hence, we do not deal with auxiliary functions in this technical report. However, since their syntax is similar to the syntax of attribute definitions, it is described in Chapter 3.

Role in Associations

An entity type that participates in an association plays a particular *role* in the relationship. EB^3 attribute definitions can use the role of an entity type in an association to determine some values in the **if** predicates. A role is not defined in EB^3 in the same way according to the association it takes part.

In associations of cardinality $M : N$, several entities of the same type can participate in the association. Since the association is defined by a recursive function that outputs the key attributes of the existing entities of the association, the key of a $M : N$ association is composed of the key attributes of the entity types that participate in the association. Let ass denote a binary association between entity types e_1 and e_2 defined by:

$$ass \ : \ \mathcal{T}(\mathsf{main}) \rightarrow \mathbb{F}(T_{k_1^1} \times \cdots \times T_{k_{m_1}^1} \times T_{k_1^2} \times \cdots \times T_{k_{m_2}^2})$$

Figure 2.1 shows the representation of roles r_1 and r_2 in association ass. The role r_1 of e_2 in ass is defined in EB³ as follows.

$$r_1(k^1_1, ..., k^1_{m_1}) = \{(k^2_1, ..., k^2_{m_2}) \mid (k^1_1, ..., k^1_{m_1}, k^2_1, ..., k^2_{m_2}) \in ass\}$$

The role r_2 of e_1 in ass is defined in EB³ as follows.

$$r_2(k^2_1, ..., k^2_{m_2}) = \{(k^1_1, ..., k^1_{m_1}) \mid (k^1_1, ..., k^1_{m_1}, k^2_1, ..., k^2_{m_2}) \in ass\}$$

Such definitions can be retrieved from the ER diagram. For instance, $membRes$ is defined by:

$$membRes(bId) = \{mId \mid (bId, mId) \in reservation\}$$

In associations of cardinality $1 : N$, the entity in the 1 side of the association is unique for each entity in the N side that participates in the association. The key of such associations can be composed by only the key attributes of the entity type in the N side of the association. Consequently, for associations of cardinality $1 : N$ (see Fig. 2.2), the role r_2 of entity type e_1 in the 1 side of the association is defined in EB³ by a recursive function of the following form:

$$r_2 \ : \ \mathcal{T}(\mathsf{main}) \times T_{k^2_1} \times \cdots \times T_{k^2_{m_2}} \rightarrow T_{k^1_1} \times \cdots \times T_{k^1_{m_1}}$$

where $T_{k^2_1}, ..., T_{k^2_{m_2}}$ denote the types of the key attributes of e_2 and $T_{k^1_1}, ..., T_{k^1_{m_1}}$ denote the types of the key attributes of e_1. Like for attribute definitions, function r_2 is total and is defined with a CAML-like pattern matching. For instance, role *borrower* in association *loan* is defined by:

$$\begin{array}{l} borrower(bId) : memberKey\_Set \ \triangleq \\ \quad \mathsf{Lend}(bId, mId, \_) \ : \ mId, \\ \quad \mathsf{Return}(bId) \ : \ \bot, \\ \quad \mathsf{Transfer}(bId, mId, \_) \ : \ mId, \\ \quad \mathsf{Take}(bId, mId, \_) \ : \ mId; \end{array}$$

Role r_1 is defined as the roles in $M : N$ associations.

Termination

Since expressions u_i of input clauses may contain a recursive call to the function, attribute definitions are recursive functions on the valid traces of the system. 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 defined by default, then computation of attribute values terminates.

Consistency Condition

We suppose that correct EB³ specifications of attribute definitions satisfy the following consistency condition: when a non-key attribute b returns a value other than $\bot$ for a key value, then the key function should contain that key.

$$\forall\, s, \overrightarrow{k} \ \bullet \ b(s, \overrightarrow{k}) \neq \bot \ \Rightarrow \ (\overrightarrow{k}) \in \kappa(s)$$

where κ is the corresponding key function. This means that the entities that are concerned by the computation of the new value of the attribute exist.

No Loop of Function Calls

Whenever expressions u_i involve other attribute definitions, but for the same valid trace s as for u_i, then we suppose that correct EB^3 specifications of attribute definitions do not contain loops of function calls between the attribute definitions.

2.6 An Example of EB^3 Attribute Definitions

Here are the attribute definitions for the library system example presented in Sect. 1.2.

Entity Type Book

$bookKey() : \mathbb{F}(bookKey\_Set) \triangleq$
  Acquire$(bId, _)$: $bookKey() \cup \{bId\}$,
  Discard(bId) : $bookKey() - \{bId\}$;

$title(bId) : Title\_Type \triangleq$
  Acquire$(bId, bTitle)$: $bTitle$,
  Discard(bId) : $\perp$,
  Modify$(bId, newTitle)$: $newTitle$;

Entity Type Member

$memberKey() : \mathbb{F}(memberKey\_Set) \triangleq$
  Register$(mId, _)$: $memberKey() \cup \{mId\}$,
  Unregister(mId) : $memberKey() - \{mId\}$;

$nbLoans(mId) : \mathbb{N} \triangleq$
  Register$(mId, _)$: 0,
  Lend$(_, mId, _)$: $1 + nbLoans(mId)$,
  Return(bId) : **if** $mId = borrower(bId)$
    then $nbLoans(mId) - 1$ **end**,
  Transfer$(bId, mId', _)$: **if** $mId = mId'$
    then $nbLoans(mId) + 1$
    else if $mId = borrower(bId)$
      then $nbLoans(mId) - 1$ **end**
    end,
  Take$(_, mId, _)$: $1 + nbLoans(mId)$,
  Unregister(mId) : $\perp$;

$loanDuration(mId) : \mathbb{N} \triangleq$
  Register(mId, lD) : lD,
  Unregister(mId) : $\perp$;

Association loan

$loan() : \mathbb{F}(bookKey\_Set) \triangleq$

$\mathsf{Lend}(bId, \_, \_)$: $loan() \cup \{bId\}$,
$\mathsf{Return}(bId)$: $loan() - \{bId\}$,
$\mathsf{Take}(bId, \_, \_)$: $loan() \cup \{bId\}$;

$dueDate(bId) : DATE \triangleq$
$\mathsf{Lend}(bId, \_, Permanent)$: $CurrentDate + 365$,
$\mathsf{Lend}(bId, mId, Classic)$: $CurrentDate + loanDuration(mId)$,
$\mathsf{Return}(bId)$: $\bot$,
$\mathsf{Transfer}(bId, mId, type)$: **if** $type = Permanent$
then $CurrentDate + 365$
else $CurrentDate + loanDuration(mId)$
end,
$\mathsf{Take}(bId, \_, Permanent)$: $CurrentDate + 365$,
$\mathsf{Take}(bId, mId, Classic)$: $CurrentDate + loanDuration(mId)$;

Association reservation

$reservation() : \mathbb{F}(bookKey\_Set \times memberKey\_Set) \triangleq$
$\mathsf{Reserve}(bId, mId)$: $reservation() \cup \{(bId, mId)\}$,
$\mathsf{Cancel}(bId, mId)$: $reservation() - \{(bId, mId)\}$,
$\mathsf{Take}(bId, mId, \_)$: $reservation() - \{(bId, mId)\}$;

$position(bId, mId) : \mathbb{N} \triangleq$
$\mathsf{Reserve}(bId, mId)$: $card(membRes(bId)) + 1$,
$\mathsf{Cancel}(bId, mId')$: **if** $mId = mId'$
then $\bot$
else if $mId \in membRes(bId) \wedge$
$position(bId, mId') < position(bId, mId)$
then $position(bId, mId) - 1$
end
end,
$\mathsf{Take}(bId, mId', \_)$: **if** $mId = mId'$
then $\bot$
else if $mId \in membRes(bId)$
then $position(bId, mId) - 1$
end
end;

Let us recall that the above-mentioned attribute definitions are correct only for the valid traces of the system. For instance, attribute *position* in association `reservation` is computed for members that have reserved a book. This definition is correct because of the following requirement: the member that can take a book when it is returned is the first member in the reservation list. Such a requirement does not appear in the attribute definition, but rather as a guard in the EB^3 process expression of association `reservation`.

Let us note that EB^3 attribute definitions can use the role of entity types in associations in the **if** predicates. There are mainly two kinds of definitions for roles in EB^3. For $M : N$ associations, roles are defined as a subset of the association entities. For instance, *membRes* is the set of members that have

reserved a book:

$$membRes(bId) = \{mId \mid (bId, mId) \in reservation\}$$

The role in the 1 side of a $1 : N$ association requires the definition of a recursive function to determine the relevant entities. For instance, role *borrower* is defined by:

$$\begin{array}{l} borrower(bId) : memberKey\_Set \triangleq \\ \quad \mathsf{Lend}(bId, mId, \_) \ : \ mId, \\ \quad \mathsf{Return}(bId) \ : \ \bot, \\ \quad \mathsf{Transfer}(bId, mId, \_) \ : \ mId, \\ \quad \mathsf{Take}(bId, mId, \_) \ : \ mId; \end{array}$$

Chapter 3

Syntax of EB^3 Attribute Definitions

3.1 Conventions

We use the BNF formalism to describe the syntax of the EB^3 language for attribute definitions. The conventions about BNF are the following ones.

- Keywords, symbols and operators of the language are quoted. For instance, " **if** ", " (" and " + ".
- Non-terminal elements of the grammar are described in italics.
- Expression e ::= *elements* denotes a definition of the grammar, where e is a non-terminal element and *elements* is one or more elements of the grammar.
- $e|e'$ denotes element e or element e'.
- $[e]$ means that element e is optional.
- (e) is equivalent to e.
- $e^\star$ denotes an arbitrary number $n \geq 0$ of elements e.
- $e^{\star\text{e'}}$ denotes n elements e, $n \geq 0$, with element e' as separator. For instance, $a^{\star\text{","}}$ stands for $a, a, ..., a$.
- e^+ and $e^{+\text{e'}}$ are similar to the definitions with $\star$, but with $n \geq 1$.

3.2 Terminal Elements

3.2.1 Characters and Strings

CHARACTER ::= "A" | ... | "Z" | "a" | ... | "z" | "0" | ... | "9"
STRING ::= CHARACTER [STRING]

3.2.2 Naturals and Integers

Number ::= “0” | ... | “9”
NATURAL ::= *Number* [NATURAL]
Sign ::= “+” | “−”
INTEGER ::= [*Sign*] NATURAL

3.2.3 Enumerated Sets

ENUMERATED-SET ::= “{” STRING$^{\star\text{“,”}}$ “}”

3.2.4 Identifiers

An identifier is simply a string without any number:

CHIdent ::= “A” | ... | “Z” | “a” | ... | “z”
STIdent ::= *CHIdent* [*STIdent*]
IDENT ::= *STIdent*

In the grammar of the language, we distinguish several kinds of identifier in order to describe for each expression which data are used and what are the links between identifiers. The following identifiers are all defined as IDENT: IdentTrace, IdentNameKey, IdentNameAttribute, IdentNameFunction, IdentParamKey, IdentNameAction, IdentParamAction, IdentVariableAction, IdentVariableIf. IdentConstant is defined as ENUMERATED-SET, NATURAL, INTEGER or STRING. In the next section, we will define for each expression what are the syntactical constraints on the idenfiers.

3.2.5 Keywords, Symbols and Operators

The list of keywords is: **match**, **with**, **if**, **then**, **else**, **end**. The list of special symbols is: $\perp$, $\_$, $\emptyset$, $\triangleq$. The list of operators is: $+$, $-$, $\times$, $/$, $\%$, $card$, $front$, $last$, $\cup$.

3.3 Syntax of Attribute Definitions

3.3.1 General Expression

$List\_AttributeDefinition$ ::= $AttributeDefinition^{\star\text{“;”}}$

$AttributeDefinition$ ::= $KeyDefinition$
| $NonKeyAttributeDefinition$
| $AuxiliaryFunction$
| $RoleAssociation$

3.3.1.1 Key Definition

KeyDefinition ::=
IdentNameKey "(" IdentTrace ") $\triangleq$"
[" **match** *last*(" IdentTrace ") **with**"]
[*Input_Clause_Key_Init* ","]
Input_Clause_Key_Update*","
[*Input_Clause_Key_RecursiveCall*]

Restriction

KeyDefinition is the general form of a key definition in EB³. IdentNameKey denotes the declaration of the name of the key. This must be the only declaration of the key in *List_AttributeDefinition*. IdentTrace in expression *last*() is the trace declared as an actual parameter of the key definition. IdentNameKey and IdentTrace, defined as IDENT, can be used in the non-terminal expressions of *KeyDefinition*. IdentNameKey can also be used in all the other non-terminal expressions of *List_AttributeDefinition*. The input clause *Input_Clause_Key_Init* is optional only for the case indicated in Sect. 3.3.2.1.

3.3.1.2 Non-Key Attribute Definition

NonKeyAttributeDefinition ::=
IdentNameAttribute "(" IdentTrace "," IdentParamKey^{+}"," ") $\triangleq$"
[" **match** *last*(" IdentTrace ") **with**"]
[*Input_Clause_Attribute_Init* ","]
Input_Clause_Attribute_Update*","
[*Input_Clause_Attribute_RecursiveCall*]

Restriction

NonKeyAttributeDefinition is the declaration of a non-key attribute definition in EB³. IdentNameAttribute denotes the unique declaration of the attribute name in *List_AttributeDefinition*. IdentTrace and the list of IdentParamKey are the formal parameters of the attribute. They can be used in the non-terminal expressions of *NonKeyAttributeDefinition*. IdentTrace in expression *last*() is the same trace as declared in the attribute definition. IdentNameAttribute can be used in all the other non-terminal expressions of *List_AttributeDefinition*. All the identifiers are defined as IDENT. The input clause *Input_Clause_Attribute_Init* is optional only for the case indicated in Sect. 3.3.2.1.

3.3.1.3 Auxiliary Functions

AuxiliaryFunction ::=
IdentNameFunction "(" IdentTrace "," IdentParamKey*"," ") $\triangleq$"
[" **match** *last*(" IdentTrace ") **with**"]
[*Input_Clause_Function_Init* ","]
Input_Clause_Function_Update*","

$[Input\_Clause\_Function\_RecursiveCall]$

Restriction

AuxiliaryFunction is the declaration of an auxiliary function in EB^3. IdentNameFunction denotes the unique declaration of the function name in the specification. IdentTrace and the list of IdentParamKey are the formal parameters of the function. They can be used in the non-terminal expressions of *AuxiliaryFunction*. IdentTrace in expression $last()$ is the same trace as declared in the auxiliary function. IdentNameFunction can be used in all the other non-terminal expressions of $List\_AttributeDefinition$. All the identifiers are defined as IDENT. The input clause $Input\_Clause\_Function\_Init$ is optional only for the case indicated in Sect. 3.3.2.1.

3.3.1.4 Roles in Associations

RoleAssociation ::=
IdentNameRole "(" IdentParamKey$^{+\text{","}}$ ") $\triangleq$"
" { " IdentParamKey$^{+\text{","}}$ " | (" IdentParamKey$^{+\text{","}}$ ") $\in$" IdentNameKey "}"

Restriction

RoleAssociation is the declaration of a role in a association in EB^3. IdentNameRole denotes the unique declaration of the role name in the specification. The list of key parameters IdentParamKey inside the parenthesis "( ) " contains all the parameters used as input parameters and in the set just after " { ". IdentNameKey corresponds to the name of an association. IdentNameRole can be used in all the other non-terminal expressions of $List\_AttributeDefinition$. All the identifiers are defined as IDENT.

3.3.2 Input Clauses for Initialization and Recursive Call

3.3.2.1 Initialization

$Input\_Clause\_Key\_Init$ ::=
"$\perp$: " $Computation\_Initialization\_Key$

$Input\_Clause\_Attribute\_Init$ ::=
"$\perp$: " $Computation\_Initialization\_Attribute$

$Input\_Clause\_Function\_Init$ ::=
"$\perp$: " $Computation\_Initialization\_Attribute$

Restriction

The input clause for initialization is optional only when:

- *Computation_Initialization_Key* is $\emptyset$, for a key.
- *Computation_Initialization_Attribute* is $\perp$, for an attribute or for an auxiliary function.

Indeed, $\emptyset$ and $\perp$ are the values by default.

3.3.2.2 Recursive Call

Input_Clause_Key_RecursiveCall ::=
“_ : ” *Recursive_Call_Same_Key*

Input_Clause_Attribute_RecursiveCall ::=
“_ : ” *Recursive_Call_Same_Attribute*

Input_Clause_Function_RecursiveCall ::=
“_ : ” *Recursive_Call_Same_Function*

3.3.3 Input Clauses for Updates

Input_Clause_Key_Update ::=
Input_Head_Key “ : ” *Functional_Term_Key*

Input_Clause_Attribute_Update ::=
Input_Head_Attribute “ : ” *Computation_Update_Attribute*

Input_Clause_Function_Update ::=
Input_Head_Function “ : ” *Computation_Update_Function*

3.3.4 Input Heads

Input_Head_Key ::=
IdentNameAction “(” (IdentVariableAction |
IdentParamAction |
“ _ ”)⋆“,” “)”

Input_Head_Attribute ::=
IdentNameAction “(” (IdentParamKey |
IdentVariableAction |
IdentParamAction |
“ _ ”)⋆“,” “)”

Input_Head_Function ::=
IdentNameAction "(" (IdentParamKey |
IdentVariableAction |
IdentParamAction |
" _ ")⋆"," ")"

Restriction

IdentNameAction corresponds to one of the EB^3 action labels that are declared in the process expression part of the EB^3 specification by:

Declaration_Action ::=
IdentNameAction "(" IdentParamAction$^{+\text{","}}$ ")"

In the above declaration, IdentNameAction and IdentParamAction are defined as IDENT and must be used in the attribute definitions.

The list of identifiers and symbols after IdentNameAction in expressions *Input_Head_Key*, *Input_Head_Attribute* and *Input_Head_Function* must be compatible with the above-mentioned declaration of the action. In particular, the number n of elements must be the same and, for each i, $i \leq n$, the i-th identifier IdentParamAction in the declaration of the action can be replaced either:

- by the same identifier IdentParamAction,
- by special symbol _,
- or by a variable IdentVariableAction

in the i-th position of the list in *Input_Head_Key*.

Moreover, for *Input_Head_Attribute* (resp. *Input_Head_Function*) expressions, IdentParamAction can also be replaced by one of the identifiers IdentParamKey declared in the attribute declaration *NonKeyAttributeDefinition* (resp. the function declaration *AuxiliaryFunction*), where the input head is defined.

IdentVariableAction is defined as IDENT and can only be used in the expression *Computation_Update_Key* or *Computation_Update_Attribute* that corresponds to the input head.

3.3.5 Expressions for Initialization

Computation_Initialization_Key ::= " ∅ "
| ENUMERATED-SET

Computation_Initialization_Attribute ::= " ⊥ "
| *Conditional_Term_Initialization_Key*

3.3.6 Expressions for Recursive Calls

Recursive_Call_Key ::=
IdentNameKey “(*front*(” IdentTrace “))”

Recursive_Call_Same_Key ::=
IdentNameKey “(*front*(” IdentTrace “))”

Recursive_Call_Attribute ::=
IdentNameAttribute “(*front*(” IdentTrace “) ,”
(IdentParamKey |
IdentConstant |
IdentVariableAction |
Computation_Expression)*“,” “)”
|
IdentNameRole “(” IdentParamKey^{+}“,” “)”

Recursive_Call_Same_Attribute ::=
IdentNameAttribute “(*front*(” IdentTrace “) ,”
(IdentParamKey |
IdentConstant |
IdentVariableAction |
Computation_Expression)*“,” “)”

Recursive_Call_Function ::=
IdentNameAttribute “(*front*(” IdentTrace “) ,”
(IdentParamKey |
IdentConstant |
IdentVariableAction |
Computation_Expression)*“,” “)”
|
IdentNameFunction “(*front*(” IdentTrace “) ,”
(IdentParamKey |
IdentConstant |
IdentVariableAction |
Computation_Expression)*“,” “)”
|
IdentNameRole “(” IdentParamKey^{+}“,” “)”

Recursive_Call_Same_Function ::=
IdentNameFunction “(*front*(” IdentTrace “) ,”
(IdentParamKey |
IdentConstant |
IdentVariableAction |
Computation_Expression)*“,” “)”

Restriction

IdentNameKey, IdentNameAttribute and IdentNameFunction must be one of the keys, attributes or auxiliary functions defined in *List_AttributeDefinition*. In the expressions with *Same*, the identifier (IdentNameKey, IdentNameAttribute and IdentNameFunction) must be the same as in the definition (of the key, attribute or auxiliary function) where the expression is used. In other words, the function defining the key, the attribute or the auxiliary function is recursively called in its own definition. IdentNameRole must be one of the roles defined in the specification.

IdentTrace in expression $front()$ is the same as in the key, the attribute or the auxiliary function definition. For *Recursive_Call_Same_Attribute* and *Recursive_Call_Same_Function*, the list of identifiers after $front$(IdentTrace) is exactly the same as in $NonKeyAttributeDefinition$ and $AuxiliaryFunction$, respectively (that is, a list of IdentParamKey). For *Recursive_Call_Function*, the list of identifiers must be compatible with the attribute definition (for the first case) or with the auxiliary function (for the second case). In particular, the number n of elements must be the same and, for each i, $i \leq n$, the i-th identifier after expression $front()$ in the recursive call must be either:

- the same identifier IdentParamKey as in $NonKeyAttributeDefinition$ (resp. $AuxiliaryFunction$),
- a constant IdentConstant from ENUMERATED-SET, INTEGER, NATURAL or STRING,
- one of the IdentVariableAction defined in the last input head analysed,
- or a computation expression using the IdentParamKey declared in attribute definition $NonKeyAttributeDefinition$ (resp. auxiliary function $AuxiliaryFunction$) and/or the IdentVariableAction defined in the last input head analysed.

3.3.7 Expressions for Updates

Computation_Update_Attribute ::= *Functional_Term_Attribute*
| *Conditional_Term_Attribute*

3.3.7.1 Functional Terms

Functional_Term_Key ::= ENUMERATED-SET
| *Recursive_Call_Same_Key* "∪" ENUMERATED-SET
| *Recursive_Call_Same_Key* "−" ENUMERATED-SET
| *Recursive_Call_Same_Key* "∪" ENUMERATED-SET
"−" ENUMERATED-SET

Functional_Term_Attribute ::= *Ground_Term_Attribute*
| *Computation_From_Attribute*

Restriction

The ground terms and the corresponding computations are defined in Sect. 3.3.9.

3.3.7.2 Conditional Terms

Conditional_Term_Initialization_Key ::=
" **if**" *Predicates_Key* " **then** "
Functional_Term_Key
[" **else** "
Conditional_Term_Initialization_Key]
" **end** "

Conditional_Term_Attribute ::=
" **if**" *Predicates_Attribute* " **then** "
Functional_Term_Attribute
[" **else** "
Conditional_Term_Attribute]
" **end** "

3.3.7.3 Updates for Auxiliary Functions

Computation_Update_Function ::= *Functional_Term_Function*
| *Conditional_Term_Function*

The functional and conditional terms for auxiliary functions are defined as for non-key attributes, but each occurrence of *Recursive_Call_Attribute* in *Computation_From_Attribute* is replaced by *Recursive_Call_Function*.

3.3.8 Predicates for Conditional Terms

Predicates_Key ::= IdentParamKey "=" IdentConstant

Predicates_Attribute ::= *Computation_Predicate*
| *Predicate_Logic*
| *Existential_Predicate*

3.3.8.1 Predicates Using Operators and Other Attributes

Computation_Predicate ::= IdentParamKey "=" *Key_From_Action*
| IdentParamKey "∈" *Set_Of_Keys_From_Action*
| *Value_From_Key* "=" *Value_From_Action*
| *Value_From_Key* "∈" *Set_Of_Values_From_Action*
| IdentParamAction "=" *Key_From_Action*
| IdentParamAction "=" *Value_From_Action*
| IdentParamAction "∈" *Set_Of_Values_From_Action*
| IdentVariableIf "=" *Value_From_Action*

| IdentVariableIf "$\in$" *Set_Of_Values_From_Action*

Restriction

IdentParamKey is an identifier defined in the attribute definition or in the auxiliary function that is analysed. IdentParamAction comes from the identifiers defined in the last input head analysed. IdentVariableIf is a variable declared in an existential predicate. The ground terms and the corresponding computations are defined in Sect. 3.3.9.

3.3.8.2 Predicate Logic

Predicate_Logic ::=
" (" *Predicates_Attribute* ") "
| "$\neg$ (" *Predicates_Attribute* ") "
| *Predicates_Attribute* "$\wedge$" *Predicates_Attribute*

Existential_Predicate ::=
" $\exists$ " IdentVariableIf " $\bullet$ " *Predicates_Attribute*

Restriction

IdentVariableIf is a fresh variable that is used in the predicate expression after symbol $\bullet$.

3.3.9 Computation Expressions

3.3.9.1 Computation Expressions in Functional Terms

Computation_From_Attribute ::= *Ground_Term_Attribute*
| *Recursive_Computation*
| *Arithmetic_Computation*
| *Set_Computation*
| *List_Computation*

Ground_Term_Attribute ::= IdentConstant
| IdentVariableAction
| IdentVariableIf
| IdentParamKey

Restriction

These expressions refer to Sect. 3.3.6 and 3.3.7. IdentConstant is a constant from ENUMERATED-SET, INTEGER, NATURAL or STRING. IdentVariableAction comes from the last input head analysed. IdentVariableIf is a fresh variable declared in one of the predicates that determine the functional term (see Sect. 3.3.8). IdentParamKey is declared in *NonKeyAttributeDefinition*.

3.3.9.2 Computation Expressions in Conditional Terms

Key_From_Action ::= *Ground_Term_Action*

Set_Of_Keys_From_Action ::= *Ground_Term_Action*
| *Recursive_Call_Key*
| *Set_Computation*

Value_From_Action ::= *Ground_Term_Action*
| *Recursive_Computation*
| *Arithmetic_Computation*

Set_Of_Values_From_Action ::= *Ground_Term_Action*
| *Recursive_Computation*
| *Set_Computation*

Value_From_Key ::= *Ground_Term_Key*
| *Recursive_Computation*
| *Arithmetic_Computation*
| *Set_Computation*

Ground_Term_Action ::= IdentConstant
| IdentParamAction
| IdentVariableAction
| IdentVariableIf
| IdentParamKey

Ground_Term_Key ::= IdentConstant
| IdentVariableIf
| IdentParamKey

Restriction

These expressions refer to Sect. 3.3.8.1. IdentConstant is a constant from ENUMERATED-SET, INTEGER, NATURAL or STRING. IdentParamAction is declared as an input parameter of the action of the input head (see Sect. 3.3.4). IdentVariableAction comes from the last input head analysed. IdentVariableIf is a fresh variable declared in one of the predicates that determine the functional term (see Sect. 3.3.8). IdentParamKey is declared in the definition of the attribute.

3.3.9.3 Common Expressions

The following are the expressions that are used in the above-mentioned computations. For the sake of concision, each occurrence of *Computation_Expression* in these expressions must be replaced by the accurate expression among:

- $Computation\_From\_Attribute$,
- $Value\_From\_Action$,
- $Set\_Of\_Values\_From\_Action$,
- Key_From_Action,
- $Set\_Of\_Keys\_From\_Action$,
- $Value\_From\_Key$.

For instance, *Computation_Expression* in *Set_Computation* must be replaced by *Computation_From_Attribute* for the expressions in Sect. 3.3.9.1.

Recursive_Computation ::= *Recursive_Call_Attribute*
| *Recursive_Call_Key*

Arithmetic_Computation ::=
Computation_Expression "+" *Computation_Expression*
| *Computation_Expression* "−" *Computation_Expression*
| *Computation_Expression* "×" *Computation_Expression*
| *Computation_Expression* "/" *Computation_Expression*
| *Computation_Expression* "%"

Set_Computation ::= "*card* (" *Computation_Expression* ") "
| *Computation_Expression* "∪" *Computation_Expression*
| *Computation_Expression* "−" *Computation_Expression*

List_Computation ::= "*front* (" *Computation_Expression* ") "
| "*last* (" *Computation_Expression* ") "

Chapter 4

Synthesizing Imperative Programs

In an EB^3 specification, there is only one state variable, the current trace of the system. Figure 4.1 shows how the IS data model is represented by the EB^3 attribute definitions from the system trace. For example, let us consider the following trace:

$$s = [\mathsf{Acquire}(b1, t1), \mathsf{Register}(m1), \mathsf{Acquire}(b2, t2),\\ \mathsf{Acquire}(b3, t3), \mathsf{Register}(m2), \mathsf{Lend}(b1, m2)]$$

The key definitions characterize the set of existing key values for each entity type and association. For instance, function $bookKey$ associates s to $\{b1, b2, b3\}$, which denotes the set of existing books. Likewise, the set of existing members is $\{m1, m2\}$ in function $memberKey$ and the set of loans includes only $(b1, m2)$ in function $loan$, since only one event of Lend occurs in trace s. The attribute values are determined by means of the non-key attribute definitions. For instance, the title of book $b2$ is computed by $title(s, b2)$.

This representation follows from the trace semantics adopted in EB^3. However, it is not appropriate to use the system trace in an *implementation* of the IS specification, because the system trace grows infinitely. Consequently, our approach is to store each attribute in a conventional relational DB which represents the ER diagram. The value of an attribute is computed when a new event is accepted by process expression *main*. In order to efficiently make these update, we must identify the subset of attributes that are affected by an action.

In this section, we show how to implement the ER diagram and the EB^3 attribute definitions. We generate, for each action a, a relational DB transaction that corresponds to the effects of a, as defined in the set of attribute definitions. In that aim, we analyse all the input clauses of EB^3 attribute definitions to determine which attributes are affected by the execution of action a and what are the effects of a on these attributes. In particular, we have to determine the key values to delete from the tables and the key values to update and/or to insert. The results of the input clauses analysis are the following: i) the attributes $B(a)$ affected by action a, ii) the tables $T(a)$ affected by a, iii) the key values $K_{Delete}(t, a)$ to delete from table t in the transaction of a, and iv) the key values $K_{Change}(t, a)$ to insert and/or to update in t. We also determine

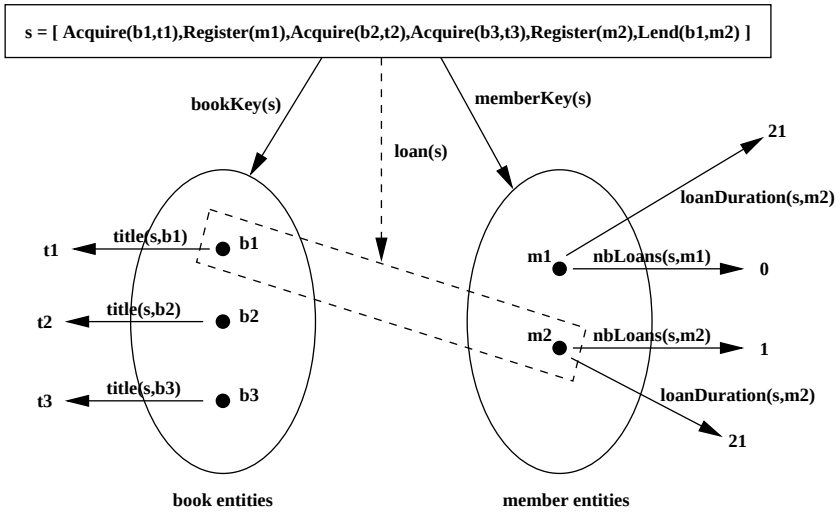

Figure 4.1: Data modelled by EB^3 attribute definitions

some temporary variables and tables for the transaction definitions in order to avoid inconsistencies because of the ordering of the different statements within each transaction. To define the transaction of a, we must generate for each table in $T(a)$ the SQL statements that correspond to the effects of a. With the sole analysis of the input clauses, one cannot distinguish the key values to insert from those to update; an analysis of EB^3 process expressions would be required for that. This issue is solved by generating a test within the transaction in order to distinguish **INSERT** from **UPDATE** statements.

create the tables of the DB
initialize the DB
for each action a of the EB^3 specification
    determine $B(a)$
    determine the temporary variables and tables (A1)
    determine $T(a)$
    for each table t in $T(a)$
        determine $K_{Delete}(t, a)$ and $K_{Change}(t, a)$
    define a transaction for a
        generate the definition of all the temporary variables and tables
        for each table t in $T(a)$
            determine and generate the SQL statements (A2)
        generate a commit

4.1 Creation of the Tables

We use standard algorithms from [Elm04] to create relational tables from the ER diagram. The signature of EB^3 actions provides the list of attributes that can be set to **NULL**. Indeed, if an input type is decorated with `^N` in the action signature, then the value of this parameter can be `NULL`. If this type appears in

the ER diagram, then the corresponding attribute can be undefined in the table. This step is detailed in [Bat05]. For instance, the table definition generated for association `loan` is:

```
CREATE TABLE loan (
    bookKey BookKey_Set PRIMARY KEY REFERENCES book,
    borrower MemberKey_Set REFERENCES member,
    dueDate INT NOT NULL
);
```

4.2 Initialization of the DB

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 $\perp : u$. 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 DB table. The most common value for u for a non-key attribute is $\perp$. This means either that there is no tuple at the initialization or that the entries are initialized to **NULL** for non-key attribute b. Otherwise, u is a functional term or a conditional term and the **INSERT** statement is generated as for the transactions in Sect. 4.4.3.

4.3 Analysis of the Input Clauses

The results of the analysis of the input clauses are the following:

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

Because of the pattern matching analysis described in Sect. 2.4, 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 $B(a)$.

For each attribute b of $B(a)$, there may be several input clauses $a(\overrightarrow{p_j}) : u_j$ with the same label a. Since the first input clause that evaluates to true is the one to be executed, analysis of these input clauses is done in their declaration order. Let $table(b)$ denote the function that returns the table where b is stored. Set $T(a)$ is then defined by:

$$T(a) = \{table(b) \mid b \in B(a)\}$$

Let t be an element of $T(a)$. To obtain $K_{Delete}(t, a)$ and $K_{Change}(t, a)$, we determine for each attribute b of $B(a)$ the corresponding sets $K_D(b)$ and $K_{IU}(b)$

such that:

$$K_{Delete}(t,a) = \bigcup_{\{b \in B(a) \wedge table(b)=t\}} K_D(b)$$

$$K_{Change}(t,a) = \bigcup_{\{b \in B(a) \wedge table(b)=t\}} K_{IU}(b)$$

$K_D(b)$ and $K_{IU}(b)$ are determined by analysing the input clauses. The subalgorithm (A1) in the general algorithm is the following:

for each attribute b of $B(a)$
  if b is defined by a conditional term
    generate a decision tree for b
    determine **SELECT** statements for the relevant records
    determine the temporary variables and the temporary tables
  determine $K_{IU}(b)$, $K_D(b)$

4.3.1 Determination of $K_{IU}(b)$ and $K_D(b)$

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 assignment. For instance, the input clause for action Transfer in attribute *nbLoans* is of the form:

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

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

Sets $K_D(b)$ are computed only for key definitions. If b is a key definition, then expression u_j is a functional term and a value v has been determined from the pattern matching. If u_j contains the symbol "$-$", then v is added to set $K_D(b)$; otherwise v is added to set $K_{IU}(b)$. For instance, the input clause of Discard in key definition *bookKey* is associated to expression $bookKey() - \{bId\}$. Consequently, we deduce that: $K_D(bookKey) = \{bId\}$.

For computing the sets $K_{IU}(b)$, we consider both key definitions and non-key attribute definitions. If expression u_j in the input clause is a functional term, then a key value v has been entirely determined. 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 $b(\texttt{trace} :: a, \overrightarrow{v}) \neq b(\texttt{trace}, \overrightarrow{v})$. 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. 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 $KV_{j,i}$. Then, each $KV_{j,i}$ is merged with $K_{IU}(b)$.

For the sake of concision, we do not deal with decision trees in this report; their construction and analysis are detailed in [GFL04]. We just illustrate this point by considering attribute definition *nbLoans*. For instance, action Transfer is associated to a conditional term in this definition: the **if** predicates

determine two key values for mId: mId' and $borrower(bId)$. In that case, $K_{IU}(nbLoans) = \{mId', borrower(bId)\}$.

4.3.2 Hypotheses on the Input Parameters

Let us identify the actual parameters of input clause $a(p_1, \ldots, p_m) : u$ with the formal parameters of action $a(q_1, \ldots, q_m)$:

$$\exists\ var(p_1, \ldots, p_m)\ \bullet \bigwedge_j p_j = q_j$$

This statement allows us to determine for which formal parameters q_j action a may be executed. There are mainly three cases to consider:

1. when p_j is the wildcard symbol '_', then u is valid for every value of parameter q_j.
2. when p_j is a variable v, then u is valid for the values of v; the pattern mapping θ_u allows us to bind variable v to parameter q_j. However, if v occurs in more than one place in $p_1, \ldots, p_m$, then we must also check that the corresponding parameters in $q_1, \ldots, q_m$ are all the same.
3. when p_j is a ground term c, then u is valid only for $q_j = c$.

Consequently, some hypotheses must be defined on the input parameters for cases 2 and 3. Moreover, other hypotheses can appear in the predicates of conditional terms.

For instance, parameter $typeOfLoan$ of action Lend$(bId, mId, typeOfLoan)$ implies two distinct input clauses in attribute definition $dueDate$. The first input clause provides a functional term for $typeOfLoan = Permanent$, while the other one is for $typeOfLoan = Classic$. Such hypotheses on the values of the input parameters imply the definition of **IF THEN ELSE END** statements in the transactions. The determination of the hypotheses and the definition of the corresponding statements are detailed in [GFL04].

4.3.3 Definition of Temporary Variables and Tables

The definition of temporary variables and tables is coupled with the analysis of the decision trees. When the records 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, such definitions allow us to define transactions independently of the statements ordering. In particular, **DELETE** statements can be grouped at the beginning of the transaction without any consequences on the other statements, since the relevant key values have been recorded in the temporary variables and/or tables. A temporary variable is defined when there is a unique record, while a temporary table is used to store several records.

For instance, a temporary variable TEMP1 is introduced as follows[1] for $borrower(bId)$ in the transaction of action Transfer:

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

VAR TEMP1 : memberKey_Set /* Define a new variable TEMP1 */
SELECT borrower **INTO** TEMP1 /* Assign the value to TEMP1 */
FROM book
WHERE bookKey = #bId;

A program variable used in an SQL statement is prefixed by the symbol "#", in order to distinguish it from attribute names. A temporary variable $TEMP$ can then be simply used in the transaction definitions by predicates of the form $param = TEMP$, where $param$ is a parameter in the **WHERE** clauses of SQL statements. A temporary table is used to characterize several key values. For instance, if we need the collection of books borrowed by member mId, then the following table is defined:

CREATE TABLE TAB1 (bookKey bookKey_Set **PRIMARY KEY**);
INSERT INTO TAB1
SELECT bookKey
FROM book
WHERE borrower = #mId;

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

4.3.4 Patterns for the SELECT Statements

The generation of **SELECT** statements that correspond to the key values satisfying the **if** predicates depends on the form of the predicate. We have identified the most typical patterns of predicates and their corresponding **SELECT** statements. 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 the sake of concision, we suppose that each recursive call $b(front(s), ...)$ is denoted by $b(...)$ in the following patterns. Let $position(p, f)$ denote the position of parameter p in function f. For instance, $position(k_3, f) = 4$ for $f(k_1, k_2, p_1, k_3)$.

4.3.4.1 Roles in Associations

A **SELECT** statement is generated for each role defined in an association. When the role is defined as a subset of the association entities, then the **SELECT** statement is directly derived from the EB^3 definition. Let ass be an association between entity types e_1 and e_2. Let $k_1^1, ..., k_{m_1}^1$ denote the key attributes that define attribute e_1, and $k_1^2, ..., k_{m_2}^2$ denote the key attributes that define attribute e_2. Let r_1 be the role of e_2 in ass defined by:

$$r_1(v_1^1, ..., v_{m_1}^1) = \{(v_1^2, ..., v_{m_2}^2) \mid (v_1^1, ..., v_{m_1}^1, v_1^2, ..., v_{m_2}^2) \in ass\}$$

The **SELECT** statement generated for r_1 is the following:

SELECT $T.k_1^2, ..., T.k_{m_2}^2$
FROM T
WHERE $T.k_1^1 = \#v_1^1$
AND ...
AND $T.k_{m_1}^1 = \#v_{m_1}^1$;

where T is the table of association ass. Likewise, if role r_2 of e_1 in ass is defined by:

$$r_2(v_1^2, ..., v_{m_2}^2) = \{(v_1^1, ..., v_{m_1}^1) \mid (v_1^1, ..., v_{m_1}^1, v_1^2, ..., v_{m_2}^2) \in ass\}$$

then the **SELECT** statement is:

SELECT $T.k_1^1, ..., T.k_{m_1}^1$
FROM T
WHERE $T.k_1^2 = \#v_1^2$
  AND ...
  AND $T.k_{m_2}^2 = \#v_{m_2}^2$;

For instance, *membRes* is defined by:

$$membRes(bId) = \{mId \mid (bId, mId) \in reservation\}$$

The generated **SELECT** statement is:

SELECT reservation.memberKey
FROM reservation
WHERE reservation.bookKey = #bId;

If the role is defined by a recursive function, because the entity particpates in a $1 : N$ association, then it does not need a **SELECT** statement, since it can be directly evaluated thanks to the recursive function. For instance, role *borrower* is implemented as an attribute of table `book` and it is taken into account in the input clauses analysis described in Sect. 4.3.

4.3.4.2 Predicates on Scalars

In this section, we consider predicates of the form $f(\overrightarrow{k}, \overrightarrow{p})$ *op* $g(\overrightarrow{k}, \overrightarrow{p})$, where f and g return scalars and *op* is a binary operator on scalars like $=$, $>$ or $<$. $\overrightarrow{k}$ is a subset of the key attributes to determine and $\overrightarrow{p}$ is a subset of the input clause parameters.

For instance, 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:

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

where m is the number of paramaters in $\overrightarrow{p}$. This pattern can be applied when the table of g has exactly m key attributes. In that case, the **SELECT** statement returns the value of attribute g for key $(p_1, ..., p_m)$.

4.3.4.2.1 Pattern $f(\vec{k}, \vec{p^1}) = g(\vec{k}, \vec{p^2})$ A more general case is a predicate expression of the form $f(\overrightarrow{k}, \overrightarrow{p^1}) = g(\overrightarrow{k}, \overrightarrow{p^2})$, where f and g are attribute recursive calls, $\overrightarrow{k} = (k_1, ..., k_n)$ is a subset of the key attributes to determine, and $\overrightarrow{p^1}$ and $\overrightarrow{p^2}$ are subsets of the input clause parameters. The input parameters of f

and g are not exactly $(\overrightarrow{k}, \overrightarrow{p^1})$ and $(\overrightarrow{k}, \overrightarrow{p^2})$, respectively. They are of the form $f(v_1, ..., v_{m_f})$ and $g(w_1, ..., w_{m_g})$, where v_i (resp. w_i) is either a key attribute k_j, or a value computed from $\overrightarrow{p^1}$ (resp. $\overrightarrow{p^2}$) with a function or an attribute definition. m_f and m_g denote the number of key attributes of $table(f)$ and $table(g)$, respectively. The **SELECT** statement is then the following.

/* extract $k_1, ..., k_n$ */
SELECT $table(f).key(position(k_1, f)), ..., table(f).key(position(k_n, f))$
FROM $table(f), table(g)$
WHERE $table(g).key(position(k_1, g)) = table(f).key(position(k_1, f))$
AND ... /* evaluation of g for each k_j, $1 \leq j \leq n$ */
AND $table(g).key(position(k_n, g)) = table(f).key(position(k_n, f))$
/* evaluation of g for w_i, $1 \leq i \leq m_g$, if w_i is not a k_j */
AND ...
AND $table(g).key(position(w_i, g)) = \#w_i$
AND ...
/* evaluation of f for v_i, $1 \leq i \leq m_f$, if v_i is not a k_j */
AND ...
AND $table(f).key(position(v_i, f)) = \#v_i$
AND ...
AND $table(f).f = table(g).g$; /* predicate */

Restriction. This pattern can be applied when $m_f \geq n$ and $m_g \geq n$. Moreover, the number of clauses of the form $table(g).key(position(w_i, g)) = \#w_i$ ($table(f).key(position(v_i, f)) = \#v_i$, respectively) should be equal to $m_g - n$ (resp. $m_f - n$). Let us note that this pattern supposes that the values of attributes f and g are scalars of the same type T. The equality $=$ can be replaced in the predicate by other binary operators op of type $T \times T$, like $>$ or $<$. In that case, the last predicate in the **WHERE** clause is replaced by $table(f).f$ op $table(g).g$. If $f = g$, then we use aliases for the table. The **FROM** clause becomes:

FROM $table(f)\ T_1, table(g)\ T_2$

Each occurrence of $table(f)$ (resp. $table(g)$) is replaced by T_1 (resp. T_2) in the **SELECT** and **WHERE** clauses of the pattern above.

4.3.4.2.2 Composition of Recursive Calls Let us now consider the simple case $f(k_1, ..., k_n) = g(p_1, ...p_m)$. If f is a composition of attributes recursive calls, then a join between the different tables whose attributes are concerned is necessary. For instance, if $f = f_1; f_2$ (*i.e.,* $f_2(f_1(k_1, ..., k_n)$ op $g(p_1, ..., p_m))$, then a new condition is added in the **SELECT** statement:

SELECT $table(f_1).key(1), ..., table(f_1).key(n)$
FROM $table(f_1), table(f_2), table(g)$
WHERE $table(f_2).key(1) = table(f_1).f_1$ /* join $f_1; f_2$ */
AND $table(g).key(1) = \#p_1$
AND ...
AND $table(g).key(m) = \#p_m$
AND $table(f_2).f_2$ op $table(g).g$;

If the composition is on the right-hand side, then the join is on g. For instance, if $g = g_1; g_2$, then the **SELECT** statement becomes:

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

Composition with Keys Defined by a Single Key Attribute. Let us now suppose that each recursive function has only one key attribute. For a predicate of the form $f_p(f_{p-1}(...f_1(k)))$ *op* S_g, where S_g is a value determined from recursive functions, the corresponding **SELECT** statement is of the following form:

SELECT $table(f_1).key(1)$ /* extract k */
FROM $table(f_1), table(f_2), ..., table(f_p)$, tables for S_g
WHERE $table(f_p).key(1) = table(f_{p-1}).f_{p-1}$ /* join $f_{p-1}; f_p$ */
  AND $table(f_{p-1}).key(1) = table(f_{p-2}); f_{p-2}$ /* join $f_{p-2}; f_{p-1}$ */
  AND ...
  AND $table(f_2).key(1) = table(f_1).f_1$ /* join $f_1; f_2$ */
  AND Conditions on S_g
  AND $table(f_p).f_p$ *op* S_g; /* predicate */

Analogously, for a predicate of the form CK *op* $g_r(g_{r-1}(...g(p)))$, where CK is a computation on k, the corresponding **SELECT** statement is of the following form:

SELECT k in the relevant table of CK
FROM tables for $CK, table(g_1), table(g_2), ..., table(g_r)$
WHERE Conditions on CK
  AND $table(g_r).key(1) = table(g_{r-1}).g_{r-1}$ /* join $g_{r-1}; g_r$ */
  AND $table(g_{r-1}).key(1) = table(g_{r-2}); g_{r-2}$ /* join $g_{r-2}; g_{r-1}$ */
  AND ...
  AND $table(g_2).key(1) = table(g_1).g_1$ /* join $g_1; g_2$ */
  AND $table(g_1).key(1) = p$
  AND CK *op* $table(g_r).g_r$; /* predicate */

Composition with Keys Defined by Several Key Attributes. When a non-key attribute h has several key attributes $k_1, ..., k_m$, then input parameter k_j of recursive function h can be instantiated by another recursive function d if the type of attribute d is the same as the type of k_j. For a composition of the form $h(..., v_{j-1}, d(...), v_{j+1}, ...)$, the corresponding predicates in the **WHERE** clause are then:

SELECT the relevant key attributes
FROM the relevant tables, including $table(h)$ and $table(d)$
WHERE ...
  /* value for the $(j-1)$-th key attribute of h */

AND $table(h).key(j-1) = v_{j-1}$
/* value for the (j)-th key attribute of h */
AND $table(h).key(j) = table(d).d$
/* value for the $(j+1)$-th key attribute of h */
AND $table(h).key(j+1) = v_{j+1}$
AND ...
AND Conditions on d
AND General predicate

4.3.4.2.3 Composition with Other Functions Recursive functions can be composed with a function using arithmetic and/or set operators. Let h and d be two attribute definitions. We suppose that h has m key attributes. Let c be a function that uses set and/or arithmetic operators. Let us now consider composition: $h(..., v_{j-1}, c(..., p_{l-1}, d(...), p_{l+1}, ...), v_{j+1}, ...)$. We suppose that it is well-defined, *i.e.,* each output type of a function given as an input parameter of another function is the type required that input parameter. Then, the pattern is the following.

SELECT the relevant key attributes
FROM the relevant tables, including $table(h)$ and $table(d)$
WHERE ...
/* value for the $(j-1)$-th key attribute of h */
AND $table(h).key(j-1) = v_{j-1}$
/* value for the j-th key attribute of h */
AND $table(h).key(j) = c(..., p_{l-1}, table(d).d, p_{l+1}, ...)$
/* value for the $(j+1)$-th key attribute of h */
AND $table(h).key(j+1) = v_{j+1}$
AND ...
AND Conditions on d
AND General predicate

If c is the last function to be composed, *e.g.,* $c(..., p_{l-1}, d(...), p_{l+1}, ...)$, then it appears in the **WHERE** clause of the predicate. For a predicate of the form $c(..., p_{l-1}, d(...), p_{l+1}, ...)$ *op* S_g, the **SELECT** statement is:

SELECT the relevant key attributes in d
FROM the relevant tables, including $table(d)$
WHERE Conditions on d
AND Conditions on S_g
AND $c(..., p_{l-1}, table(d).d, p_{l+1}, ...)$ *op* S_g; /* predicate */

For a predicate of the form CK *op* $c(..., p_{l-1}, d(...), p_{l+1}, ...)$, the **SELECT** statement is of the following form:

SELECT the relevant key attributes of CK
FROM the relevant tables, including $table(d)$
WHERE Conditions on CK
AND Conditions on d
AND CK *op* $c(..., p_{l-1}, table(d).d, p_{l+1}, ...)$; /* predicate */

4.3.4.3 Patterns on Sets

In this section, we consider predicates of the form $f(\overrightarrow{k}, \overrightarrow{p})$ *ops* $g(\overrightarrow{k}, \overrightarrow{p})$, where f and g return sets and *ops* is a binary operator on sets like $=$ or $\subseteq$. $\overrightarrow{k}$ is a subset of the key attributes to determine and $\overrightarrow{p}$ is a subset of the input clause parameters. Since recursive functions defining attributes output only scalars, then there are only two cases where f or g can output sets. Either the set of input parameters is a subset of the key of the attribute, or one the input parameter is a set of possible values. In the former case, f or g then corresponds to a role and such patterns are defined in Sect. 4.3.4.1. In the latter case, expressions $f(\overrightarrow{k}, \overrightarrow{p})$ and $g(\overrightarrow{k}, \overrightarrow{p})$ then correspond to the range of f and g.

4.3.4.3.1 Predicate of the Form $k \in g(\vec{p})$ Let us consider a predicate of the form $k \in g(\vec{p})$, 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:

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

where m is the number of paramaters in $\overrightarrow{p}$. This pattern can be applied when the table of g has more than m key attributes. In that case, the **SELECT** statement returns the values of attribute g for the subset of key $(p_1, ..., p_m)$. This case corresponds to the pattern for a role; g is indeed a role for the association defined in $table(g)$.

4.3.4.3.2 Range of Attribute Definitions Let us now consider the simple case $k \in g(p_1, ..., S_i, ...p_m)$, where S_i is a set of possible values for input parameter p_i. By our algorithm, set S_i is itself determined by a **SELECT** pattern defined as a temporary table TAB in the host language. Then recursive function g outputs one attribute value for each element in TAB. To avoid confusion, parentheses are now replaced by brackets in expression $g(p_1, ..., S_i, ...p_m)$ to point out that the range of g is considered instead of a single output value: $g[\ p_1, ..., S_i, ...p_m]$. The **SELECT** statement is then of the following form.

SELECT $table(g).g$
FROM $table(g)$
WHERE $table(g).key(1) = \#p_1$
  AND ...
  AND $table(g).key(i)$ **IN** (**SELECT** $TAB.p_i$ **FROM** TAB)
  AND ...
  AND $table(g).key(m) = \#p_m$;

4.3.4.3.3 Composition of Ranges Let us now suppose that the predicate of the form is of the form $k \in f[...p_{i-1}, g[...], p_{i+1}...]$, where f is an attribute definitions, $p_1, ..., p_{i-1}, p_{i+1}, ..., p_m$ denote input parameters and $g[...]$ is the range of attribute definition g. Then a **SELECT** statement can be generated for expression $g[...]$, as described in Sect. 4.3.4.3.2. Let TAB denote the temporary

table defined from the **SELECT** statement. The **SELECT** statement for the composition is then of the following form:

SELECT $table(f).f$
FROM $table(f)$
WHERE $table(f).key(1) = \#p_1$
  AND ...
  AND $table(f).key(i-1) = \#p_{i-1}$
  AND $table(f).key(i)$ **IN** (**SELECT** $TAB.p_i$ **FROM** TAB)
  AND $table(f).key(i+1) = \#p_{i+1}$
  AND ...
  AND $table(f).key(m) = \#p_m$;

4.3.4.3.4 Predicate of the Form $f[\vec{k}, \vec{p^1}] \subseteq g[\vec{k}, \vec{p^2}]$ In that case, at least one the input parameters in $\vec{p^1}$ and one of the input parameters in $\vec{p^2}$ are sets of values; for example, input parameters $p^1_{j_1}$ and $p^2_{j_2}$ are sets $S^1_{j_1}$ and $S^2_{j_2}$, respectively. Let $TAB1$ and $TAB2$ denote the temporary tables determined for $S^1_{j_1}$ and $S^2_{j_2}$. The **SELECT** statement for the predicate is then of the following form.

/* extract $k_1, ..., k_n$ */
SELECT $table(f).key(position(k_1, f)), ..., table(f).key(position(k_n, f))$
FROM $table(f)$ F_1, $table(g)$ G_1
WHERE $G_1.key(position(k_1, g)) = F_1.key(position(k_1, f))$
  AND ... /* evaluation of g for each k_j, $1 \leq j \leq n$ */
  AND $G_1.key(position(k_n, g)) = F_1.key(position(k_n, f))$
  /* evaluation of g for each p^1_i, if p^1_i is neither a k_j nor a set */
  AND ...
  AND $G_1.key(position(p^1_i, g)) = \#p^1_i$
  AND ...
  /* evaluation of g for $S^1_{j_1}$ */
  AND $G_1.key(j_2)$ **IN** (**SELECT** $TAB2.p_{j_2}$ **FROM** $TAB2$)
  /* evaluation of f for each p^2_i, if p^2_i is neither a k_j nor a set */
  AND ...
  AND $F_1.key(position(p^2_i, f)) = \#p^2_i$
  AND ...
  /* evaluation of f for $S^1_{j_1}$ */
  AND $F_1.key(j_1)$ **IN** (**SELECT** $TAB1.p_{j_1}$ **FROM** $TAB1$)
  /* inclusion of sets */
  AND NOT EXISTS
    (**SELECT** $F_2.f$
    FROM $table(f)$ F_2
    WHERE $F_2.key(1) = F_1.key(1)$
      AND...
      AND $F_2.key(m_f) = F_1.key(m_f)$
      AND NOT EXISTS
        (**SELECT** $G_2.g$
        FROM $table(g)$ G_2
        WHERE $G_2.key(1) = G_1.key(1)$
          AND...

AND $G_2.key(m_g) = G_1.key(m_g)$
AND $G_2.g = F_2.f)$);

where m_f and m_g are the number of key attributes of $table(f)$ and $table(g)$, respectively.

4.3.4.4 Composition of Patterns

During the analysis of the input clauses of an attribute definition, a temporary table is generated for each leaf of the decision tree to record the key values that are affected by the action. The table is defined from a **SELECT** statement that is the resulting composition of several **SELECT** patterns. Indeed, each leaf of the decision tree is associated to the conjunction of conditions labelling the path's edges leading to the leaf. For each condition, we can use some of the **SELECT** patterns described in the previous sections. The patterns are then composed between them according to their logical connection. The algorithm is the following.

```
for each decision tree
    determine the key K of the attribute
    determine the set of paths that lead to a leaf, except the last one
    for each path
        for each condition c_i labelling the path's edges
            if c_i is a positive predicate
                for each conjunct c_{i,j} in c_i
                    if c_{i,j} is a positive condition
                        apply the relevant SELECT patterns for c_{i,j} (S1)
                    else
                        apply the relevant SELECT patterns for ¬ c_{i,j} (S1)
                        define a temporary table for ¬ c_{i,j}
                define a SELECT statement for K (S2)
            else
                find the SELECT statement already defined for ¬ c_i (S3)
                define a temporary table for ¬ c_i (S4)
        determine the SELECT statement for the path (S5)
        define the temporary table for the path
```

Steps (S1)-(S5) are detailed in the next paragraphs.

4.3.4.4.1 Application of Patterns Step (S1) corresponds to the application of the patterns described in Sect. 4.3.4.2 and 4.3.4.3. If predicate $c_{i,j}$ involves a comparison between scalars, then Sect. 4.3.4.2 provides the elementary patterns and explains how to build the **SELECT** statement if some attributes are composed in the predicate. If predicate $c_{i,j}$ involves roles or ranges of attributes, then Sect. 4.3.4.3 provides the elementary patterns and explains how to build the **SELECT** statement when ranges of attributes are composed.

4.3.4.4.2 Definition of the Statement for c_i Step (S2) allows the **SELECT** statement generated from patterns to be intersected with the other statements. Predicate $c_{i,j}$ can involve only a subset of the key K that must

be determined. To consider the intersection of the different **SELECT** statements coming from the different conditions of c_i, we use the Cartesian product to take the missing key attributes into account. Thus, a **SELECT** statement of the following form

SELECT $T.k_1, ..., T.k_n$
FROM T
WHERE /* predicates */

is transformed into a statement of the following form:

SELECT $t1.k_1, ..., t1.k_n, t2.l_1, ..., t2.l_m$
FROM T $t1, T$ $t2$
WHERE /* predicates where each occurrence of T is replaced by $t1$ */

where $l_1, ..., l_m$ are the key attributes of K that do not appear in predicate $c_{i,j}$.

Once a statement has been computed for each $c_{i,j}$ in c_i, then a global statement is defined for c_i. Let us recall that a **SELECT** statement is generated for the positive condition of each $c_{i,j}$; hence, if $c_{i,j}$ is a negative condition, then the statement has been generated for $\neg\ c_{i,j}$. The algorithm is the following.

determine the list l of conditions $c_{i,j}$ in c_i
determine the list lp of positive conditions in c_i
let $lw = []$ be a list of conditions
let $ls = []$ be a list of statements
for each condition $c_{i,j}$ in l
  if $c_{i,j}$ is in lp
    for each $c_{i,j'}$ in lw
      add condition K **NOT IN** (**SELECT** $\star$ **FROM** TAB),
        where TAB is the temporary table associated to $c_{i,j'}$,
        in the **WHERE** clause of the **SELECT** statement
        associated to $c_{i,j}$
      remove $c_{i,j'}$ from lw
      add the **SELECT** statement into ls
  else
    add $c_{i,j}$ into lw
if lw is not empty
  define a **SELECT** statement of the following form: (S6)
    SELECT K
    FROM t;
    where t is the table in the **FROM** clause of
    the temporary table defined for the first condition in lw
  for each $c_{i,j'}$ in lw
    add condition K **NOT IN** (**SELECT** $\star$ **FROM** TAB),
      where TAB is the temporary table associated to $c_{i,j'}$,
      in the **WHERE** clause of the **SELECT** statement
      defined at step (S6)
    remove $c_{i,j'}$ from lw
    add the **SELECT** statement into ls
define the temporary table for c_i as the intersection of the
  SELECT statements in ls

Consequently, the resulting table has the following form.

```
/* creation of table with the relevant key attributes */
CREATE TABLE TAB (...);
INSERT INTO TAB
/* First statement in ls */
    SELECT ...
    FROM ...
    WHERE ...;
    INTERSECT
    ...
    INTERSECT
/* Last statement in ls */
    SELECT ...
    FROM ...
    WHERE ...;
```

4.3.4.4.3 Reuse of Statements A decision tree is a binary tree in DNF (see Sect. 4.3.1 and [GFL04]); hence, each condition labelling a right edge is the negation of the condition labelling the left edge. If condition c_i is a negative predicate (*e.g.,* of the form $\neg(...)$), then it corresponds to the condition labelling the right edge. Since the decision tree is analysed by a depth-first traversal in prefix order, the **SELECT** statement that corresponds to the left edge has already been generated for $\neg c_i$. In step (S3), this statement is retrieved and a temporary table is defined from it in step (S4), as described in Sect. 4.3.3.

4.3.4.4.4 Definition of the Statement for the Path Once all the conditions labelling the path's edges have been analysed, a global statement is generated for the path. The conditions labelling the path's edges are divided into two groups: the positive and the negative ones. For each positive predicate, a **SELECT** statement has been generated (step (S2)). For each negative predicate, a temporary table has been defined (step (S4)). The algorithm to generate the global statement is the following.

```
determine the list L of predicates labelling the path's edges
determine the list LP of positive predicates
let LW = [] be a list of predicates
let LS = [] be a list of statements
for each predicate c_i in L
    if c_i is in LP
        for each c_j in LW
            add condition K NOT IN (SELECT * FROM TAB),
                where TAB is the temporary table associated to c_j,
                in the WHERE clause of the SELECT statement
                associated to c_i
            remove c_j from LW
            add the SELECT statement into LS
    else
        add c_i into LW
```

define the temporary table as the intersection of the
    SELECT statements in LS

Consequently, the resulting table has the following form.

```
/* creation of table with the relevant key attributes */
CREATE TABLE TAB (...);
INSERT INTO TAB
/* First statement in LS */
    SELECT ...
    FROM ...
    WHERE ...;
    INTERSECT
    ...
    INTERSECT
/* Last statement in LS */
    SELECT ...
    FROM ...
    WHERE ...;
```

4.4 Definition of Transactions

For defining transactions, all the SQL 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 are in set $K_{Delete}(t, a)$. The **DELETE** statements are grouped at the beginning of each table's list of instructions. A test is defined to determine whether the key values in $K_{Change}(t, a)$ already exist in the tables, in order to distinguish updates from insertions. In that aim, we define L as the list of potential **UPDATE** statements for table t in the transaction corresponding to action `a`. We first try the first update of L. Then, we test whether the key value has not been found in the DB. If so, then the **INSERT** statement is executed instead; otherwise, the other updates of L are executed. The subalgorithm (A2) of the general algorithm is the following:

```
for each k in K_Delete(t, a)
    determine and generate the DELETE statement with k
for each k in K_Change(t, a)
    L := []
    for each attribute b of t in B(a)
        if k is in K_IU(b)
            compute the value of b(k)
            determine the UPDATE statement for b(k)
            insert the UPDATE statement into L
    define S_I as the INSERT statement for k and the b(k)s
    define S_U as the sequence of the updates of front(L)
    generate the first update of L
    generate the following statement:
        IF SQL%NotFound
```

THEN S_I
ELSE S_U
END;

4.4.1 DELETE statements

If $K_{Delete}(t, a)$ is simply a singleton of the form $\{\overrightarrow{k}\}$, where $\overrightarrow{k} = (k_1, ..., k_m)$ and $k_1, ..., k_m$ are value constants, then the corresponding **DELETE** statement is of the following form:

DELETE FROM t
WHERE $key(1) = \#k_1$
  AND ...
  AND $key(m) = \#k_m$;

More generally, $K_{Delete}(t, a)$ is a set of key values. Some of them are directly determined, while others are characterized by temporary variables and/or tables. By definition,

$$K_{Delete}(t, a) = \bigcup_{\{b \in B(a) \wedge t = table(b)\}} K_D(b)$$

So, $K_{Delete}(t, a)$ can be considered as a set of the form $\{SKD_1, ..., SKD_n\}$, where the SKD_is are the sets $K_D(b)$ such that $t = table(b)$. Each SKD_i, $1 \leq i \leq n$, is a set of key values (possibly a singleton) characterized either by temporary variables or tables, or by key values. A **DELETE** statement of the following form is generated for each SKD_i:

DELETE FROM t
WHERE (WHD);

If SKD_i is determined from a temporary variable $TEMP$, then (WHD) is of the form: $(key(1), ..., key(m)) = TEMP$. If SKD_i is determined from a temporary table TAB, then (WHD) is: $(key(1), ..., key(m))$ **IN** (**SELECT** $key(1), ..., key(m)$ **FROM** TAB). If SKD_i is a singleton $\{kv\}$, then (WHD) is: $(key(1), ..., key(m)) = \#kv$.

4.4.2 UPDATE statements

Let us recall that:

$$K_{Change}(t, a) = \bigcup_{\{b \in B(a) \wedge table(b) = t\}} K_{IU}(b)$$

$K_{Change}(t, a)$ can be considered as the set of the form $\{SKC_1, ..., SKC_n\}$, where the SKC_is are the sets $K_{IU}(b)$ such that $t = table(b)$. Each SKC_i, $1 \leq i \leq n$, is either a singleton or a set of key values. The key values of each SKC_i are then grouped by their effect on b; in other words, some of them are characterized by the same temporary table, because their update is computed with the same functional term in b.

SKC_i can be considered as the set of the form $\{SKV_{i,1}, ..., SKV_{i,l}\}$, where each $SKV_{i,j}$, $1 \leq j \leq l$, is the set of key values that are associated to the same

functional term $u_{i,j}$ in the attribute definition of b. Consequently, for each b in $B(a)$ such that $table(b) = t$, and for each subset $SKV_{i,j}$, a **UPDATE** statement of the following form is generated:

UPDATE t
SET $b = u_{i,j}$
WHERE (WHU);

If $SKV_{i,j}$ is determined from a temporary variable $TEMP$, then (WHU) is of the form: $(key(1), ..., key(m)) = TEMP$. If $SKV_{i,j}$ is determined from a temporary table TAB, then (WHU) is: $(key(1), ..., key(m))$ **IN** (**SELECT** $key(1), ..., key(m)$ **FROM** TAB). If $SKV_{i,j}$ is a singleton $\{kv\}$, then (WHU) is: $(key(1), ..., key(m)) = \#kv$.

4.4.3 INSERT statements

Like in Sect. 4.4.2, the key values of each SKC_i are grouped by their effect on b. For each subset $SKV_{i,j}$ of key values that have the same functional term $u_{i,j}$ in attribute definition b, we compute the corresponding values with $u_{i,j}$. So, for each key value kv in $K_{Change}(t, a)$, we are able to determine a new value for each b in $B(a)$ such that $table(b) = t$. For each kv, a **INSERT** statement of the following form is generated:

INSERT INTO $t(k, b_1, ..., b_r)$
VALUES $(kv, b_1(kv), ..., b_r(kv))$;

where k is the key of t, and $b_1, ..., b_r$ are the attributes of t in $B(a)$.

When the key values kv are stored in a temporary table TAB, a cursor C is required for generating a **INSERT** statement for each key value in TAB.

DECLARE C **CURSOR**
FOR
  SELECT k
  FROM TAB;
OPEN C;
VAR TEMP : T_k /* type of k */
WHILE /* end of cursor */ **DO**
  FETCH C **INTO** TEMP;
  ... /* definition of temporary variables */
  INSERT INTO $t(k, b_1, ..., b_r)$
  VALUES $(\text{TEMP}, b_1(\text{TEMP}), ..., b_r(\text{TEMP}))$;
END;
CLOSE C;

where k is the key of t, and $b_1, ..., b_r$ are the attributes of t in $B(a)$. Let us note that the computation of $b_1(\text{TEMP}), ..., b_r(\text{TEMP})$ may require the definition of temporary variables.

4.4.4 IF statements

To distinguish updates from insertions in $K_{Change}(t, a)$, a test is defined for each key value of $K_{Change}(t, a)$. Let us note that several updates can be generated,

while only one insertion is required for the same key value. Indeed, for each key value kv in $K_{Change}(t, a)$, one can compute a new attribute value $b(k)$ for each b in $B(a)$ such that $table(b) = t$ and determine, for each b, a **UPDATE** statement as described in Sect. 4.4.2. Let $L = [UPD_1, ..., UPD_n]$ be the list of updates generated for kv. For the same key value kv, one can determine a **INSERT** statement, denoted by INS, as described in Sect. 4.4.3. Then, for each kv, the following statement is generated to distinguish updates from insertions:

UPD_1;
IF SQL%NotFound
THEN INS
ELSE
  UPD_2;
  ...
  UPD_n
END;

Predicate SQL%NotFound is true if key value kv indicated in the **WHERE** clause of UPD_1 has not been found in the DB.

4.5 Example

The programs generated for the library management system by the algorithms presented in this chapter are the following ones:

```
CREATE TABLE book (
  bookKey bookKey_Set PRIMARY KEY,
  title CHAR(30)
);

CREATE TABLE member (
  memberKey memberKey_Set PRIMARY KEY,
  nbLoans INT NOT NULL,
  loanDuration INT NOT NULL
);

CREATE TABLE loan (
  bookKey bookKey_Set PRIMARY KEY REFERENCES book,
  dueDate INT NOT NULL,
  borrower memberKey_Set REFERENCES member
);

CREATE TABLE reservation (
  bookKey bookKey_Set REFERENCES book,
  memberKey memberKey_Set REFERENCES member,
  position INT NOT NULL,
  PRIMARY KEY (bookKey,memberKey)
);

TRANSACTION Acquire(bId : bookKey_Set, bTitle : CHAR(30))
```

```
    UPDATE book SET title = #bTitle
    WHERE bookKey = #bId;
    IF SQL%NotFound
    THEN
        INSERT INTO book(bookKey,title)
        VALUES (#bId,#bTitle);
    END;
    COMMIT;

TRANSACTION Discard(bId : bookKey_Set)
    DELETE FROM book
    WHERE bookKey = #bId;
    COMMIT;

TRANSACTION Modify(bId : bookKey_Set, nTitle : CHAR(30))
    UPDATE book SET title = #nTitle
    WHERE bookKey = #bId;
    IF SQL%NotFound
    THEN
        INSERT INTO book(bookKey,title)
        VALUES (#bId,#nTitle);
    END;
    COMMIT;

TRANSACTION Register(mId : memberKey_Set, lD : INT)
    UPDATE member SET nbLoans = 0
    WHERE memberKey = #mId;
    IF SQL%NotFound
    THEN
        INSERT INTO member(memberKey,nbLoans,loanDuration)
        VALUES (#mId,0,#lD);
    ELSE
        UPDATE member SET loanDuration = #lD
        WHERE memberKey = #mId;
    END;
    COMMIT;

TRANSACTION Unregister(mId : memberKey_Set)
    DELETE FROM member
    WHERE memberKey = #mId;
    COMMIT;

TRANSACTION Lend(bId : bookKey_Set, mId : memberKey_Set,
            typeOfLoan : Loan_Type)
    VAR TEMP1 : INT
        SELECT nbLoans + 1 INTO TEMP1
        FROM member
        WHERE memberKey = #mId;
    VAR TEMP2 : INT
        SELECT CurrentDate + loanDuration INTO TEMP2
```

```
        FROM member
        WHERE memberKey = #mId;
    UPDATE member SET nbLoans = nbLoans + 1
    WHERE memberKey = #mId;
    IF SQL%NotFound
    THEN
        INSERT INTO member(memberKey,nbLoans)
        VALUES (#mId,TEMP1);
    END;
    IF typeOfLoan = Permanent
    THEN
        UPDATE loan SET dueDate = CurrentDate + 365
        WHERE bookKey = #bId;
        IF SQL%NotFound
        THEN
            INSERT INTO loan(bookKey,dueDate,borrower)
            VALUES (#bId,CurrentDate + 365,#mId);
        ELSE
            UPDATE loan SET borrower = #mId
            WHERE bookKey = #bId;
        END;
    ELSE
        UPDATE loan SET dueDate = TEMP2
        WHERE bookKey = #bId;
        IF SQL%NotFound
        THEN
            INSERT INTO loan(bookKey,dueDate,borrower)
            VALUES (#bId,TEMP2,#mId);
        ELSE
            UPDATE loan SET borrower = #mId
            WHERE bookKey = #bId;
        END;
    END;
    COMMIT;

TRANSACTION Return(bId : bookKey_Set)
    VAR TEMP1 : memberKey_Set
        SELECT borrower INTO TEMP1
        FROM loan
        WHERE bookKey = #bId;
    VAR TEMP2 : INT
        SELECT nbLoans − 1 INTO TEMP2
        FROM member
        WHERE member = TEMP1;
    DELETE FROM book
    WHERE bookKey = #bId;
    UPDATE member SET nbLoans = nbLoans − 1
    WHERE memberKey = TEMP1;
    IF SQL%NotFound
    THEN
```

```
        INSERT INTO member(memberKey,nbLoans)
        VALUES (TEMP1,TEMP2);
    END;
    COMMIT;

TRANSACTION Transfer(bId : bookKey_Set, mId : memberKey_Set,
            typeOfLoan : Loan_Type)
    VAR TEMP1 : INT
        SELECT nbLoans + 1 INTO TEMP1
        FROM member
        WHERE memberKey = #mId;
    VAR TEMP2 : memberKey_Set
        SELECT borrower INTO TEMP2
        FROM loan
        WHERE bookKey = #bId;
    VAR TEMP3 : INT
        SELECT nbLoans − 1 INTO TEMP3
        FROM member
        WHERE member = TEMP2;
    VAR TEMP4 : INT
        SELECT CurrentDate + loanDuration INTO TEMP4
        FROM member
        WHERE memberKey = #mId;
    UPDATE member SET nbLoans = nbLoans + 1
    WHERE memberKey = #mId;
    IF SQL%NotFound
    THEN
        INSERT INTO member(memberKey,nbLoans)
        VALUES (#mId,TEMP1);
    END;
    UPDATE member SET nbLoans = nbLoans − 1
    WHERE memberKey = TEMP2;
    IF SQL%NotFound
    THEN
        INSERT INTO member(memberKey,nbLoans)
        VALUES (TEMP2,TEMP3);
    END;
    IF typeOfLoan = Permanent
    THEN
        UPDATE loan SET dueDate = CurrentDate + 365
        WHERE bookKey = #bId;
        IF SQL%NotFound
        THEN
            INSERT INTO loan(bookKey,dueDate,borrower)
            VALUES (#bId,CurrentDate + 365,#mId);
        ELSE
            UPDATE loan SET borrower = #mId
            WHERE bookKey = #bId;
        END;
    ELSE
```

```
        UPDATE loan SET dueDate = TEMP4
        WHERE bookKey = #bId;
        IF SQL%NotFound
        THEN
            INSERT INTO loan(bookKey,dueDate,borrower)
            VALUES (#bId,TEMP4,#mId);
        ELSE
            UPDATE loan SET borrower = #mId
            WHERE bookKey = #bId;
        END;
    END;
    COMMIT;

TRANSACTION Reserve(bId : bookKey_Set, mId : memberKey_Set)
    VAR TEMP1 : INT
        SELECT COUNT(memberKey) + 1 INTO TEMP1
        FROM reservation
        WHERE bookKey = #bId;
    UPDATE reservation SET position = TEMP1
    WHERE (bookKey,memberKey) = (#bId,#mId);
    IF SQL%NotFound
    THEN
        INSERT INTO reservation(bookKey,memberKey,position)
        VALUES (#bId,#mId,TEMP1);
    END;
    COMMIT;

TRANSACTION Take(bId : bookKey_Set, mId : memberKey_Set,
            typeOfLoan : Loan_Type)
    VAR TEMP1 : INT
        SELECT nbLoans + 1 INTO TEMP1
        FROM member
        WHERE member = #mId;
    VAR TEMP2 : INT
        SELECT CurrentDate + loanDuration INTO TEMP2
        FROM member
        WHERE memberKey = #mId;
    CREATE TABLE TAB1 (memberKey memberKey_Set PRIMARY KEY);
    INSERT INTO TAB1
        SELECT memberKey
        FROM reservation
        WHERE bookKey = #bId;
    UPDATE member SET nbLoans = nbLoans + 1
    WHERE memberKey = #mId;
    IF SQL%NotFound
    THEN
        INSERT INTO member(memberKey,nbLoans)
        VALUES (#mId,TEMP1);
    END;
    IF typeOfLoan = Permanent
```

```
    THEN
        UPDATE loan SET dueDate = CurrentDate + 365
        WHERE bookKey = #bId;
        IF SQL%NotFound
        THEN
            INSERT INTO loan(bookKey,dueDate,borrower)
            VALUES (#bId,CurrentDate + 365,#mId);
        ELSE
            UPDATE loan SET borrower = #mId
            WHERE bookKey = #bId;
        END;
    ELSE
        UPDATE loan SET dueDate = TEMP2
        WHERE bookKey = #bId;
        IF SQL%NotFound
        THEN
            INSERT INTO loan(bookKey,dueDate,borrower)
            VALUES (#bId,TEMP2,#mId);
        ELSE
            UPDATE loan SET borrower = #mId
            WHERE bookKey = #bId;
        END;
    END;
    DELETE FROM reservation
    WHERE bookKey = #bId AND memberKey = #mId;
    DECLARE C1 CURSOR
    FOR
        SELECT memberKey
        FROM TAB1;
    OPEN C1;
    VAR TEMP3 : memberKey_Set;
    WHILE /* end of cursor */ DO
        FETCH C1 INTO TEMP3;
        VAR TEMP4 : INT
            SELECT position − 1 INTO TEMP4
            FROM reservation
            WHERE memberKey = TEMP3;
        UPDATE reservation SET position = position − 1
        WHERE bookKey = #bId AND memberKey = TEMP3;
        IF SQL%NotFound
        THEN
            INSERT INTO reservation(bookKey,memberKey,position)
            VALUES (#bId,TEMP3,TEMP4);
        END;
    END;
    CLOSE C1;
    COMMIT;

TRANSACTION Cancel(bId : bookKey_Set, mId : memberKey_Set)
    VAR TEMP1 : INT
```

```
    SELECT position INTO TEMP1
    FROM reservation
    WHERE memberKey = #mId AND bookKey = #bId;
CREATE TABLE TAB1 (memberKey memberKey_Set PRIMARY KEY);
INSERT INTO TAB1
    SELECT memberKey
    FROM reservation
    WHERE bookKey = #bId;
CREATE TABLE TAB2 (memberKey memberKey_Set PRIMARY KEY);
INSERT INTO TAB2
    SELECT memberKey
    FROM reservation
    WHERE position > TEMP1 AND bookKey = #bId;
CREATE TABLE TAB3 (memberKey memberKey_Set PRIMARY KEY);
INSERT INTO TAB3
    (SELECT memberKey FROM TAB1)
    INTERSECT
    (SELECT memberKey FROM TAB2);
DELETE FROM reservation
WHERE bookKey = #bId AND memberKey = #mId;
DECLARE C1 CURSOR
FOR
    SELECT memberKey
    FROM TAB3;
OPEN C1;
VAR TEMP2 : memberKey_Set;
WHILE /* end of cursor */ DO
    FETCH C1 INTO TEMP2;
    VAR TEMP3 : INT
        SELECT position − 1 INTO TEMP3
        FROM reservation
        WHERE memberKey = TEMP2;
    UPDATE reservation SET position = position − 1
    WHERE bookKey = #bId AND memberKey = TEMP2;
    IF SQL%NotFound
    THEN
        INSERT INTO reservation(bookKey,memberKey,position)
        VALUES (#bId,TEMP2,TEMP3);
    END;
END;
CLOSE C1;
COMMIT;
```

4.6 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 $B(a)$, 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 a symbol "$\cup$" 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.

4.6.1 Examples

For instance, by applying the optimization for action Transfer, the results are the following ones. The attributes affected by action Transfer are *dueDate* and *borrower* in table loan and *nbLoans* in table member, with:

$$K_{Change}(dueDate) = K_{Change}(borrower) = \{bId\}$$

$$K_{Change}(nbLoans) = \{mId', borrower(bId)\}$$

Neither *memberKey* nor *loan* includes an input clause with a symbol "$\cup$" for action Transfer. Consequently, the SQL statements are updates:

TRANSACTION Transfer(bId : bookKey_Set, mId : memberKey_Set,
typeOfLoan : Loan_Type)
VAR TEMP1 : memberKey_Set
SELECT borrower **INTO** TEMP1
FROM loan
WHERE bookKey = #bId;
VAR TEMP2 : INT
SELECT CurrentDate + loanDuration **INTO** TEMP2
FROM member
WHERE memberKey = #mId;
UPDATE member **SET** nbLoans = nbLoans + 1
WHERE memberKey = #mId;
UPDATE member **SET** nbLoans = nbLoans − 1
WHERE memberKey = TEMP1;
UPDATE loan **SET** borrower = #mId
WHERE bookKey = #bId;
IF typeOfLoan = Permanent
THEN
UPDATE loan **SET** dueDate = CurrentDate + 365
WHERE bookKey = #bId;
ELSE
UPDATE loan **SET** dueDate = TEMP2
WHERE bookKey = #bId;
END;
COMMIT;

Thanks to this optimization, we avoid the definition of four **IF** statements and four insertions in transaction Transfer. We can also optimize the transactions corresponding to actions Modify, Lend, Take and Cancel in the same way.

4.6.2 Limits

Nevertheless, the analysis of key definitions is not sufficient to avoid the other **IF** statements in the transactions above. Moreover, a transaction like Acquire (see Sect. 4.5) cannot be optimized, because the input clause of Acquire in key definition *bookKey* contains symbol ∪. The synthesis of relational DB transactions presented in this technical report is planned to be coupled later with the interpretation of EB³ process expressions.

By interpreting process expressions, we will be able to optimize the transactions obtained in this report. For instance, the **producer-modifier-consumer** pattern [FSD03] is a usual pattern for entity type process expressions in EB³. 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.

Chapter 5

Tool EB^{3}TG

The APIS project [FFLR02] aims at synthesizing IS directly and automatically from EB^3 specifications. Figure 5.1 represents the main components of the APIS project. The IS implementation is obtained in APIS by interpretation and synthesis of/from the different parts of an EB^3 specification. A first tool, called DCI-Web, allows us to generate Web interfaces from GUI specifications [Ter05]. To query and/or to update the system, an IS end-user generates an event through the Web interface. This event is then analysed by EB^{3}PAI, an interpreter for EB^3 process expressions [FF02]. If it is considered as valid by the interpreter, then the event is executed; otherwise, an error message is sent to the user.

The algorithms presented in this report have been implemented in a new tool, called EB^{3}TG, to actually "execute" each valid event. In EB^3, the DB is represented by the ER diagram and by the attribute definitions. In the implementation, we do not keep track of the trace, because it would be inefficient to store the system trace; we rather store the current values of each attribute for the current trace. Thus, EB^{3}TG automatically generates, for each EB^3 action, a Java program that executes a relational DB transaction. The synthesized transactions implement the specification of IS attributes in EB^3. They can be used by EB^{3}PAI to query and/or to update the DB when the corresponding events are considered as valid by interpretation of EB^3 process expressions. The tool EB^{3}TG also generates Java programs that correspond to the creation and the initialization of the DB. Last but not least, several DB management systems (DBMS), like Oracle, PostgreSQL and MySQL, are supported by EB^{3}TG.

5.1 Description of EB^{3}TG

The tool has been implemented in Java. The code includes 50 classes, 625 methods and 20 KLOCs. The functional architecture and the various inputs and outputs of EB^{3}TG are described in Fig. 5.2.

An XML description of the EB^3 diagram is checked by EB^{3}TG with respect to the document type definition (DTD) of the ER model. Error messages are returned in case of problems. The tool then generates a relational DB schema from the XML description. The SQL statements are synthesized following the DBMS chosen by the user. The current version of EB^{3}TG supports Oracle, PostgreSQL and MySQL. For instance, the DB schema generated for the library manage-

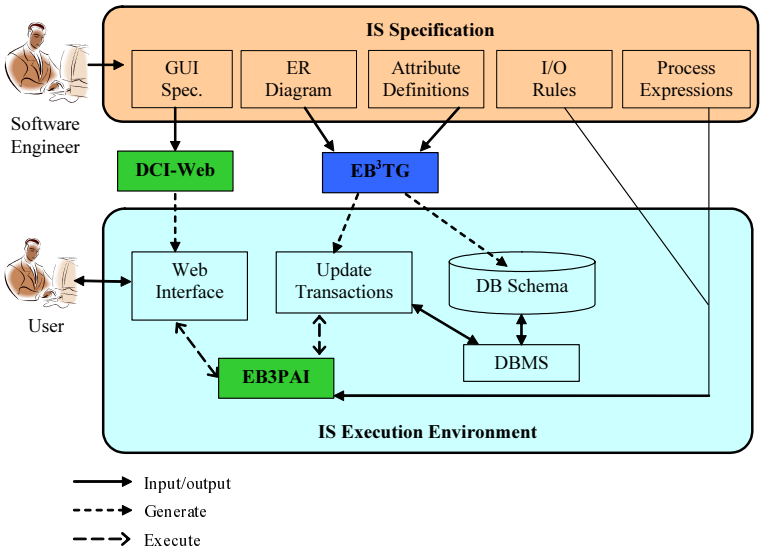

Figure 5.1: Components of the APIS project

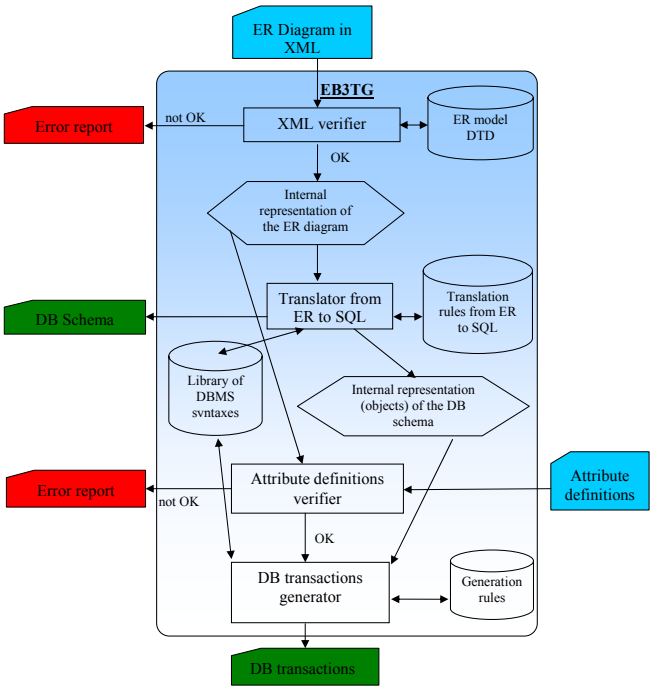

Figure 5.2: Functional architecture of EB^{3}TG

ment system is presented in Fig. 5.3. A table is created for each entity type and association of the system. Referential constraints are also automatically generated at the end to deal with mutual references between tables. For this example, Oracle is the chosen DBMS.

EB³TG also checks that attribute definitions are consistent with respect to the ER diagram. For instance, Fig. 5.4 shows two examples of syntax errors. In the first example, keyword `match` is missing at column 9, line 40 of file `bookStore.txt` where the attribute definitions are described. The second error message points out that the number of parameters of recursive call `loanDuration` does not correspond to the number of parameters in its definition. This error is in the input clause associated to action `Lend` of attribute definition `loan.dueDate`.

Finally, EB³TG synthesizes the Java programs that execute relational DB transactions corresponding to EB³ attribute definitions. For instance, the effect of action Transfer(bId, mId) is to transfer the loan of book bId to member mId. The Java method generated by EB³TG for this action is represented in Fig. 5.5. The JDBC (Java DB Connectivity) technology allows Java programs to access the DBMS. Two classes of the JDBC programming interface may execute SQL statements to update and/or to query DB: *PrepareStatement* and *Statement*. The former is more efficient in time since SQL queries are compiled only once at the beginning of the execution, but class *Statement* is implemented by every DBMS. For the sake of portability, we have chosen to use the latter class. Method *createStatement()* creates a new object of class *Statement*, while methods *executeUpdate(query)* and *executeQuery(query)* respectively execute update and query SQL statements. The use of method *executeUpdate* is illustrated in lines 29, 32, 36 and 42, in Fig. 5.5.

In order to keep track of the results of **SELECT** statements, we use the class *ResultSet*, because the objects of this class are not altered by subsequent updates. For instance, the analysis of attribute *nbLoans* requires the construction of a decision tree. In lines 8 and 14, $rset0$ and $rset1$ respectively store the results of the **SELECT** statements associated to the first and the second leaf of this decision tree. They are later used in lines 35 and 41 to update the number of loans of the previous and the new borrower of book bId. In that case, a `while` loop is generated since the result of a **SELECT** statement can be a bag of values.

5.2 Strengths and Weaknesses

The synthesized programs introduce some overhead, because they systematically store the current values of attributes before updating the DB, in order to ensure correctness. We plan to optimize these programs by analysing dependencies between update statements and avoid, when possible, these intermediate steps. By focusing on the translation of attribute definitions, the resulting transactions do not take the behaviour specified by the EB³ process expression into account, which also introduces some overhead by requiring an update/insert combination, instead of simply doing an insert. This work must now be coupled with the analysis and/or the interpretation of EB³ process expressions.

The tool has the advantage of ensuring that concurrency is properly handled in the IS by using the appropriate transaction isolation level. It currently uses

```
CREATE TABLE book (
bookKey numeric(5,2),
title varchar(20),
CONSTRAINT PKbook PRIMARY KEY(bookKey)
);

CREATE TABLE member (
memberKey numeric(5),
nbLoans numeric(5) NOT NULL,
loanDuration numeric(3) NOT NULL,
CONSTRAINT PKmember PRIMARY KEY(memberKey)
);

CREATE TABLE loan (
borrower numeric(5),
bookKey numeric(5,2),
dueDate date,
CONSTRAINT PKloan PRIMARY KEY(bookKey)
);

CREATE TABLE reservation (
bookKey numeric(5,2),
memberKey numeric(5),
position numeric(5),
CONSTRAINT PKreservation PRIMARY KEY(bookKey,memberKey)
);

ALTER TABLE loan ADD CONSTRAINT FKloan_member FOREIGN KEY (borrower)
REFERENCES member (memberKey) INITIALLY DEFERRED;

ALTER TABLE loan ADD CONSTRAINT FKloan_book FOREIGN KEY (bookKey)
REFERENCES book (bookKey) INITIALLY DEFERRED;

ALTER TABLE reservation ADD CONSTRAINT FKreservation_book FOREIGN KEY (bookKey)
REFERENCES book (bookKey) INITIALLY DEFERRED;

ALTER TABLE reservation ADD CONSTRAINT FKreservation_member FOREIGN KEY (memberKey)
REFERENCES member (memberKey) INITIALLY DEFERRED;
```

Figure 5.3: DB schema generated for the library

```
bookStore.txt:40:9:  expecting "with", found 'NULL'

>>Error in :
Attribute definition :  loan.dueDate
Action :  Lend(_,mId)
Cause :  Invalid number of parameters in attribute call
Clues :  The attribute call 'member.loanDuration'
must have exactly 2 parameters
```

Figure 5.4: Two examples of error messages

```
1     public static void Transfer(int bId,int mId){
2       try {
3         connection.createStatement().executeUpdate(
4             "CREATE TABLE eb3Tempmember ( "memberKey  numeric(5))");
5         connection.createStatement().executeUpdate(
6             "INSERT INTO  eb3Tempmember (memberKey) values("+mId+")");
7
8         ResultSet rset0 =connection.createStatement().
9           executeQuery("SELECT C.memberKey,A.nbLoans+1 "+
10          "FROM eb3Tempmember C,member A "+
11          "WHERE C.memberKey = "+mId+" "+
12          "AND A.memberKey = C.mId ");
13
14        ResultSet rset1 = connection.createStatement().
15          executeQuery("SELECT G.borrower,E.nbLoans-1 "+
16          "FROM loan G,member E "+
17          "WHERE G.bookKey = "+bId+" "+
18          "AND G.borrower NOT IN ( "+
19          "SELECT C.memberKey "+
20          "FROM eb3Tempmember C "+
21          "WHERE C.memberKey = "+mId+" ) "+
22          "AND E.memberKey = G.borrower ");
23
24        ResultSet rset2 = connection.createStatement().
25          executeQuery("SELECT D.loanDuration "+
26          "FROM member D WHERE D.memberKey = "+mId+" ");
27        String var0 = ((rset2.next())?rset2.getDouble(1)+"":"null");
28
29        connection.createStatement().executeUpdate("UPDATE loan SET "+
30          "borrower = "+mId+" WHERE bookKey = "+ bId +" ");
31
32        connection.createStatement().executeUpdate("UPDATE loan SET "+
33          "dueDate = SYSDATE+"+var0+" WHERE bookKey = "+ bId +" ");
34
35        while(rset0.next()) {
36          connection.createStatement().executeUpdate(
37          "UPDATE member SET nbLoans = "+rset0.getDouble(2)+ " "+
38          "WHERE memberKey = "+ rset0.getDouble(1));
39        }
40
41        while(rset1.next()) {
42          connection.createStatement().executeUpdate(
43          "UPDATE member SET nbLoans = "+rset1.getDouble(2)+ " "+
44          "WHERE memberKey = "+ rset1.getDouble(1));
45        }
46
47        connection.createStatement().
48          executeUpdate("DROP TABLE eb3Tempmember");
49        connection.commit();
50      } catch ( Exception e ) {
51        try{
52          connection.createStatement().
53          executeUpdate("DROP TABLE   eb3Tempmember");
54          connection.rollback();
55        } catch (SQLException s){ System.err.println(s.getMessage());}
56         System.err.println(e.getMessage());}
57    }
```

Figure 5.5: Java method for action Transfer

TRANSACTION_SERIALIZABLE, the safest mode, but also the most expensive in computational time. However, it could be extended to implement more efficient concurrency control techniques like the two-phase locking protocol, since we can identify all the tuples that have to be updated in a transaction, and use a global ordering on the tables to issue **SELECT FOR UPDATE NOWAIT** (in Oracle syntax for instance) to lock the tuples.

This tool also simplifies software maintenance. For instance, imagine that an attribute must be renamed or moved from an entity type to an association. In EB^3, only the specification is modified; the Java programs that update the DB can then be regenerated automatically.

Chapter 6

Conclusion

EB^3 is radically different from the paradigms widely used for specifying IS. An IS is considered as a black box and the specification points out the valid system inputs and outputs. Thus, the dynamic of the data model is described by means of recursive functions on the IS valid input traces. The contributions presented in this report address the automatic generation of relational DB transactions that correspond to the EB^3 attribute definitions. We have introduced an algorithm that synthesizes relational DB transactions from EB^3 attribute definitions. Synthesized programs can be used in concrete implementations of EB^3 specifications; their algorithmic complexity is similar to those of manually written programs. We have also described the tool EB^{3}TG that implements the synthesis of relational DB transactions.

Our programs introduce some overhead, because they systematically store the current values of attributes before updating the DB, 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 programs. The current values of the system trace and of the EB^3 attributes are represented by the current state of the DB. Hence, synthesized programs are efficient in terms of space complexity.

Several papers deal with the synthesis of relational implementations. Most of the time, refinement techniques are used, like in [Edm95] for Z and [Mam02] for B specifications, which are orthogonal in specification style to EB^3 [FFLar]. There exist some tools for synthesizing systems by interpretation and/or simulation [GS90, LN99], but they are generally inefficient for IS.

Concerning the synthesis of **SELECT** statements from the first-order predicates of the conditional terms, our work is close to Hohenstein's SQL/EER language [HE92]. SQL/EER is a formal query language that can be automatically translated into SQL [Hoh89]. Contrary to SQL/EER, that is independent of other relational DB specification languages, the EB^3 language for attribute definition is part of the EB^3 method, and consequently, the synthesis of imperative programs is tightly coupled with the interpretation of EB^3 process expressions. In [LT94], a formal query language is defined for EER schemas. Queries use predicates similar to our conditional terms, but the purpose is for deductive DB.

Object-relational mapping tools like Hibernate [JBo06] are remotely connected to EB^{3}TG, in the sense that they both use a relational DB representation. However, these tools support the persistence of objects of an OO programming language using a relational DB, while EB^{3}TG supports the persistence of EB^3 attribute definitions, which are more abstract. The query language supported by these tools is closer to SQL than EB^3 attribute definitions.

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 [FFLar]. 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 [GFL04], we use the B language [Abr96] to represent EB^3 attribute definitions and to verify safety properties.

With the implementation of EB^{3}TG, the main components of APIS are now ready to be related to each other. In particular, the interpreter needs to take the computation of attribute values into account in order to evaluate action guards of EB^3 process expressions. Moreover, we plan to combine the generation of transactions with the interpretation of EB^3 process expressions in order to optimize the resulting programs.

Bibliography

[Abr96] J.R. Abrial. *The B-Book: Assigning programs to meanings*. Cambridge University Press, 1996.

[Bat05] P. Batanado. Synthèse de transactions de base de données relationnelle à partir de définitions d'attributs EB^3. Master's thesis, Département d'informatique, Université de Sherbrooke, 2005.

[CM98] G. Cousineau and M. Mauny. *The functional approach to programming*. Cambridge University Press, Cambridge, 1998.

[Edm95] D. Edmond. Refining database systems. In *Proc. ZUM'95*, LNCS, Limerick, Ireland, 1995. Springer-Verlag.

[Elm04] R. Elmasri. *Fundamentals of Database Systems*. Addison-Wesley, 2004.

[FF02] B. Fraikin and M. Frappier. EB3PAI: An interpreter for the EB^3 specification language. In *15th International Conference on Software and Systems Engineering and their Applications (ICSSEA)*, December 2002.

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

[FFLR02] M. Frappier, B. Fraikin, R. Laleau, and M. Richard. APIS — Automatic Production of Information Systems. In *AAAI Spring Symposium*, pages 17–24, Stanford, USA, March 2002. AAAI Press.

[FSD03] 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.

[GFL04] F. Gervais, M. Frappier, and R. Laleau. *Synthesizing B substitutions for EB^3 attribute definitions*. Technical Report 683, CEDRIC, Paris, France, November 2004.

[GS90] 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.

[HE92] 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.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hoh89] 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.

[JBo06] JBoss Labs. Hibernate. `http://www.hibernate.org`, 2006.

[LN99] 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.

[LT94] M.J. Lawley and R.W. Topor. A query language for EER schemas. In *Proc. 5th Australian Database Conf.*, Christchurch, New Zealand, 1994.

[Mam02] A. Mammar. *Un environnement formel pour le développement d'applications base de données*. PhD thesis, CNAM, 2002.

[Ngu98] H.P. Nguyen. *Dérivation de spécifications formelles B à partir de spécifications semi-formelles*. PhD thesis, CNAM, 1998.

[Ter05] J.-G. Terrillon. Description comportementale d'interface web. Master's thesis, Département d'informatique, Université de Sherbrooke, 2005.