EB3TG: A TOOL SYNTHESIZING RELATIONAL DATABASE TRANSACTIONS FROM EB3 ATTRIBUTE DEFINITIONS

Frédéric Gervais
CEDRIC, CNAM-IIE
18 Allée Jean Rostand, 91025 Evry, France

Panawé Batanado, Marc Frappier
GRIL, Université de Sherbrooke
Sherbrooke (Québec) J1K 2R1, Canada

Régine Laleau
LACL, Université Paris 12
IUT Fontainebleau, 77300 Fontainebleau, France

Keywords: EB3, trace, pattern matching, transaction, Java, SQL.

Abstract: EB3 is a formal language for specifying information systems (IS). In EB3, the sequences of events accepted by the system are described with a process algebra; they represent the valid traces of the IS. Entity type and association attributes are computed by means of recursive functions defined on the valid traces of the system. In this paper, we present EB3TG, a tool that synthesizes Java programs that execute relational database transactions which correspond to EB3 attribute definitions.

1 INTRODUCTION

We are mainly interested in the formal specification of information systems (IS) (Gervais, 2004). In our viewpoint, an IS is a software system that allows an organization to collect and to manipulate all its relevant data. In particular, an IS includes software applications and tools to query and modify the 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 (Frappier and St-Denis, 2003; Mammar, 2002) is justified by the high value of data from corporations like banks, insurance companies, high-tech industries or government organizations.

Currently, the most widely used paradigm for specifying IS is the state transition paradigm. In state-based specifications, a system is generally described by defining state invariant properties that must be preserved by the execution of operations. Thus, data integrity constraints are described by means of invariant properties. For instance, existing approaches using state transitions for specifying IS include RoZ (Dupuy et al., 2000), OMT-B (Meyer and Souquières, 1999) and UML-B (Laleau and Mammar, 2000). EB^3 (Frappier and St-Denis, 2003) is a formal language specially created for specifying IS. The language is based on event traces and the approach is orthogonal in specification style with respect to formal languages based on state transitions (Fraikin et al., 2005).

1.1 An Overview of EB3

EB^3 (entity-based black box) is a formal language inspired from the JSD (Jackson system development) method (Jackson, 1983) and from the black box concept of Cleanroom (Prowell et al., 1999). A *black box* is a function from sequences of input events to outputs. The terms *entity type* and *entity* are used instead of class and object, respectively. In EB^3, a process algebra inspired from CSP (Hoare, 1985) and LOTOS (Bolognesi and Brinksma, 1987), is used to specify IS entities as black boxes. Thus, the sequences of events accepted by the IS, which are called the *valid input traces* of the system, are described by process expressions.

The core of EB^3 includes a process and a formal notation to described a precise and complete specification of the input-output behaviour of IS. An EB^3 specification is composed of five parts:

1. A user requirements class diagram describes the entity types and associations of the IS, and their

Gervais F., Batanado P., Frappier M. and Laleau R. (2006).
EB3TG: A TOOL SYNTHESIZING RELATIONAL DATABASE TRANSACTIONS FROM EB3 ATTRIBUTE DEFINITIONS.
In *Proceedings of the Eighth International Conference on Enterprise Information Systems - ISAS*, pages 44-51
DOI: 10.5220/0002494800440051

respective actions and attributes. This diagram is based on entity-relationship (ER) model concepts (Elmasri and Navathe, 2004) and uses a UML-like graphical notation.

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 Web interfaces used to interact with IS end-users.

The current trace of the system is the finite list of input events accepted and executed by the system; it is denoted by `trace`. In EB^3, an event is an instance of an action (*i.e.*, the execution of an action). Let $t :: \sigma$ denote the right append of an input event σ to trace t, and let `[]` denote the empty trace. The behaviour of the IS is defined as follows.

```
trace := [];
forever do
    receive input event $\sigma$;
    if main can accept trace :: $\sigma$ then
        trace := trace :: $\sigma$;
        send output event $o$ following $R$;
    else
        send error message;
```

A complete description of EB^3 can be found in (Frappier and St-Denis, 2003).

1.2 The APIS Project

The APIS project (Frappier et al., 2002) aims at synthesizing IS directly from EB^3 specifications. Figure 1 represents the different components of the APIS project. Rather than using refinement techniques to implement the system like in state-based formal languages (Edmond, 1995; Mammar, 2002), the IS is interpreted and/or synthesized from the different components of an EB^3 specification. A first tool, called DCI-Web, allows us to generate Web interfaces from GUI specifications (Terrillon, 2005). 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 (Fraikin and Frappier, 2002). If it is considered as valid by the interpreter, then the event is executed; otherwise, an error message is sent to the user.

In EB^3, the DB is represented by the user requirements class diagram and by the attribute definitions. In this paper, we present EB^{3}TG, a new tool for APIS that automatically generates, for each EB^3 action, a Java program that executes a relational DB transaction. The synthesized transactions correspond to the specification of IS attributes in EB^3. Hence, 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. Several DB management systems (DBMS), like Oracle, PostgreSQL and MySQL, are supported by EB^{3}TG.

Section 2 briefly introduces the EB^3 attribute definitions. In Sect. 3, we present the algorithms for synthesizing relational DB transactions that correspond to EB^3 attribute definitions and the EB^{3}TG tool. We conclude the paper with some perspectives for the tool and the APIS project in Sect. 4.

2 EB3 ATTRIBUTE DEFINITIONS

In this section, we first introduce an example that will be used in the remainder of the paper and then we present the EB^3 attribute definitions.

2.1 Example

To illustrate the main aspects of this paper, an example of a library management system is introduced. The system has to manage book loans by members. A book is acquired by the library; it can be discarded, but only if it is not borrowed. A member must join the library in order to borrow a book and he can relinquish library membership only when all his loans are returned or transferred. A member can also transfer a loan to another member. A book can be borrowed by only one member at once. Figure 2 represents the user requirements class diagram for the library. The process expression and the input-output rules of this example can be found in (Gervais et al., 2004).

2.2 Attribute Definitions

The definition of an attribute in EB^3 is a recursive function on the valid traces of the system, that is, the traces accepted by process expression `main`. The function is total and is given in a functional style, as in CAML (Cousineau and Mauny, 1998). It outputs the attribute values that are valid for the state in which the system is, after having executed the input events in the trace.

We distinguish key attributes from non-key attributes. A key definition outputs the set of existing key values, while a non-key attribute definition outputs the attribute value for a key value given as

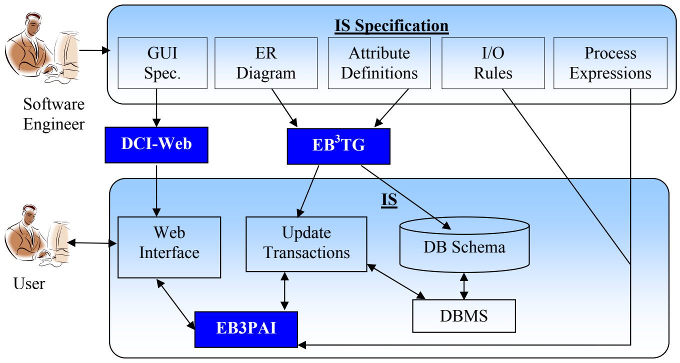

Figure 1: Components of the APIS project.

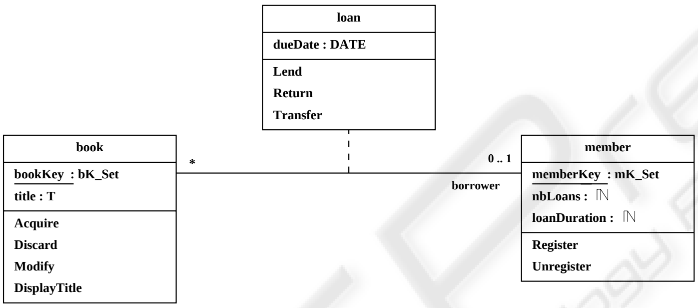

Figure 2: EB^3 specification: user requirements class diagram for the library.

an input parameter. For instance, the key of entity type `book` is defined by function $bookKey$ in Fig. 3. $bookKey$ has a unique input parameter $s \in \mathcal{T}(\text{main})$, *i.e.*, a valid trace of the system, and it returns the set of key values of entity type `book`. Let us note that type $\mathbb{F}(bK\_Set)$ denotes the set of finite subsets of bK_Set. Non-key attributes $title$ and $nbLoans$ are defined in Fig. 3. For the sake of concision, the definition of $nbLoans$ is truncated; only the effect of action Transfer is kept to illustrate the contribution of the paper. Expressions of the form $input : expr$, like $\mathsf{Acquire}(bId, ttl) : ttl$ in $title$, are called *input clauses*.

When an attribute definition is executed, then all the input clauses of the attribute definition are analysed, and the first pattern matching that holds is the one executed. Hence, the ordering of the input clauses is important. The pattern matching analysis always involves the last input event of trace s. If one of the expressions $input$ matches with the last event of trace s, denoted by $last(s)$, then the corresponding expression $expr$ is computed; otherwise, the function is recursively called with $front(s)$, that is, s truncated by removing its last element. This case corresponds to the last input clause with symbol '_'. EB^3 attribute definitions always include $\bot$, that matches with the empty trace, to represent undefinedness; hence, EB^3 recursive functions are always total. Any reference to a key $eKey$ or to an attribute b in an input clause is always of the form $eKey(front(s))$ or $b(front(s), \ldots)$. For instance, we have the following values for attribute $title$:

$$title([\,], b_1) \overset{(I1)}{=} \bot$$
$$title([\mathsf{Register}(m_1)], b_1) \overset{(I5)}{=} title([\,], b_1) \overset{(I1)}{=} \bot$$
$$title([\mathsf{Register}(m_1), \mathsf{Acquire}(b_1, t_1)], b_1) \overset{(I2)}{=} t_1$$

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 clause matches Register, and then (I1). In the last case, the value is obtained directly from (I2).

Expression $expr$ in an input clause of the form $input : expr$ is a term composed of constants, variables and attribute recursive calls. **if then else end** expressions are also used when the pattern matching condition is not sufficient. For instance, the input clause for Transfer in attribute $nbLoans$ is represented in Fig. 3. An expression $expr$ without any condition is called a *functional term*, while an expression of the **if then else end** form is a *conditional term*.

A more detailed description of EB^3 attribute definitions can be found in (Gervais et al., 2005a). The at-

$bookKey(s : \mathcal{T}(\mathsf{main})) : \mathbb{F}(bK\_Set) \triangleq$
match $last(s)$ **with**
$\bot : \emptyset,$
$\mathsf{Acquire}(bId, \_) : bookKey(front(s)) \cup \{bId\},$
$\mathsf{Discard}(bId) : bookKey(front(s)) - \{bId\},$
$\_ : bookKey(front(s));$

$title(s : \mathcal{T}(\mathsf{main}), bId : bK\_Set) : T \triangleq$
match $last(s)$ **with**
$\bot : \bot,$ (I1)
$\mathsf{Acquire}(bId, ttl) : ttl,$ (I2)
$\mathsf{Discard}(bId) : \bot,$ (I3)
$\mathsf{Modify}(bId, ttl) : ttl,$ (I4)
$\_ : title(front(s), bId);$ (I5)

$nbLoans(s : \mathcal{T}(\mathsf{main}), mId : mK\_Set) : \mathbb{N} \triangleq$
match $last(s)$ **with**
  $\bot : \bot,$
  ...
  $\mathsf{Transfer}(bId, mId') :$ **if** $mId = mId'$ **then** $nbLoans(front(s), mId) + 1$
    else if $mId = borrower(front(s), bId)$
      then $nbLoans(front(s), mId) - 1$ **end**
    end,
  ...
  $\_ : nbLoans(front(s), mId);$

Figure 3: Examples of EB^3 attribute definitions.

tribute definitions of the example are in (Gervais et al., 2004).

3 EB3TG

In this section, we introduce the main algorithm for synthesizing relational DB transactions that correspond to EB^3 attribute definitions and we present the EB^{3}TG tool.

3.1 Main Algorithm

EB^3 attribute definitions describe the dynamic behaviour of IS data. We have chosen to implement them by a relational DB. The DB allows us to store the current value of each attribute, in other words, the value for the current trace of the system. This choice avoids keeping track of the system trace, which would be difficult because of its increasing size. A relational DB transaction is generated for each action in the user requirements class diagram. Thus, every time an event is considered as valid by the interpreter, then the corresponding transaction updates the attributes affected by the action.

To generate a program that executes a relational DB transaction associated with an action a, we must analyse the input clauses of the attribute definitions in order to determine: i) which attributes are affected by the execution of action a, ii) which tables of the DB are affected by a, and iii) what are the effects of a on the attributes. We note $Att(a)$ the set of attributes affected by a and $T(a)$ the set of tables affected by a. The main algorithm is the following.

```
(1)  translate the class diagram
         into a relational DB schema
(2)  for each EB$^3$ action $a$
(3)      analyse the input clauses
(4)          determine $Att(a)$
(5)          determine $T(a)$
(6)          for each table $t$ in $T(a)$
(7)              determine the key values
                     to delete
(8)              determine the key values
                     to insert and/or to update
(9)      define the transaction for $a$
```

The different steps of the algorithm are only summed up in this paper; they are detailed in (Gervais et al., 2004; Gervais et al., 2005b; Gervais et al., 2005a).

Step (1). The DB is generated from the EB^3 specification. We use standard algorithms from (Elmasri and Navathe, 2004) to translate the user requirements class diagram into a relational DB schema (Batanado, 2005).

Steps (4)-(5). To execute an attribute definition, all the input clauses are analysed, and the first input clause that matches with the last event of the trace is the one executed. Consequently, an attribute is affected by action a if there exists at least an input clause of the form $a(\overrightarrow{p}) : expr$ in its definition. To compute $T(a)$, a function $table$, generated in step (1), associates each attribute to its table in the DB. Hence, $T(a) = \{table(b) \mid b \in Att(a)\}$.

Steps (7)-(8). To execute an attribute definition, if an input clause matches with the last event of the trace, then an assignment of a value for each free variable in the input clause has been determined. For

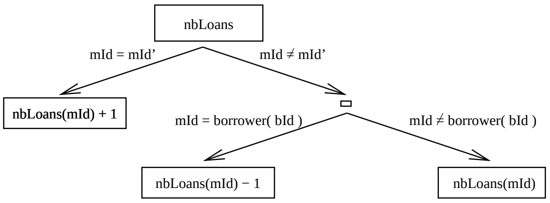

Figure 4: Decision tree for input clause Transfer in definition $nbLoans$.

instance, let suppose that Acquire$(number, title)$ is the last valid event of the IS. In that case, to execute $title$ (see Fig. 3), input clause Acquire$(bId, _)$ matches with the event and free variable bId of the attribute definition is bound to value $number$. Hence, the value of the key of $title$ has been entirely determined.

Nevertheless, when expression $expr$ in an input clause of the form $input\ :\ expr$ contains **if then else** expressions, then we must analyse the different conditions in the **if** predicates to determine the values of the key attributes that are not bound by the pattern matching. We use binary trees called *decision trees* to analyse the **if** predicates; their construction and analysis are detailed in (Gervais et al., 2004). For instance, the **if** predicates in the conditional term of input clause Transfer in $nbLoans$ determine two key values for mId: mId' and $borrower(bId)$. Figure 4 shows the decision tree obtained for this input clause. For the sake of concision, expression $front(s)$ has been removed from the attribute recursive calls. The first leaf corresponds to condition $mId = mId'$, and the second leaf to condition $mId \neq mId' \wedge mId = borrower(bId)$. The last leaf is the recursive call of $nbLoans$ from the input clause with symbol '_'.

SELECT statements are then generated from the decision trees in order to characterize the key values to delete, update or insert. Moreover, they allow us to define transactions independently of the statements ordering. We have identified the most typical patterns of predicates and their corresponding **SELECT** statements in (Gervais et al., 2005b). The key values to delete are in expressions $expr$ of the form $eKey(front(s)) - \{k\}$ in the input clauses of key definitions. A **DELETE** statement is then generated. The other key values correspond either to insertions or updates. We cannot distinguish key values to insert from key values to update at this step, since in expressions $expr$ of the form $eKey(front(s)) \cup S$, sets $eKey(front(s))$ and S are not necessarily disjoint. Note that an expression $f(front(s))$ always refers to the current value of attribute f, *i.e.*, its value before the update.

Step (9). The ordering of SQL statements in the generated transactions is the following.

1. list of the **SELECT** statements identified by the analysis of the input clauses,
2. list of **DELETE** statements,
3. list of SQL statements for insertions and/or updates.

We use a high-level pseudo-code to describe the synthesized transactions; this pseudo-code is translated into Java in EB^{3}TG. The transaction generated for action Discard is:

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

Let us note that this transaction should be executed only when Discard is a valid input event of the system. When the action involves updates and/or insertions, then the transaction becomes more complex. Indeed, tests must be defined to determine whether the key values already exist in the tables, in order to distinguish updates from insertions. For instance, the transaction generated for Acquire is:

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

The variable "SQL%NotFound" contains a value returned by the DBMS to determine whether the update has been successful.

3.2 EB3TG

The tool has been implemented in Java. The code includes 50 classes, 625 methods and 20 KLOCs. The functional architecture and the various input/output of EB^{3}TG are described in Fig. 5.

An XML description of the user requirements class diagram is first 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

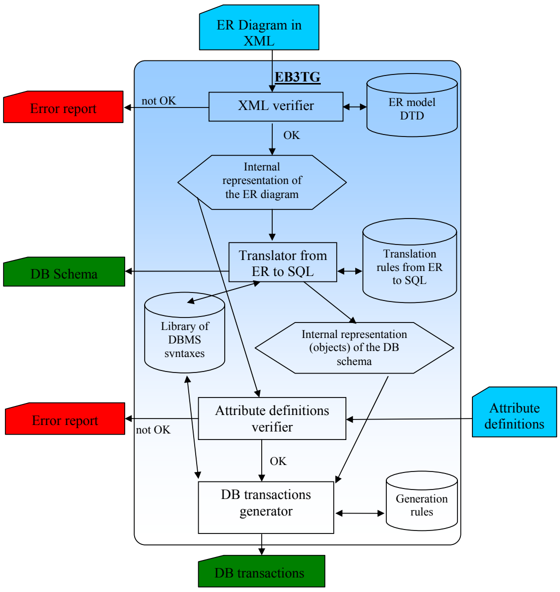

Figure 5: Functional behaviour of EB^{3}TG.

generated for the library management system is presented in Fig. 6. 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.

EB^{3}TG also checks that attribute definitions are consistent with respect to the class diagram. For instance, Fig. 7 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 `member.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, EB^{3}TG synthesizes the Java programs that execute relational DB transactions corresponding to EB^3 attribute definitions. For instance, the effect of Transfer(bId, mId) is to transfer the loan of book bId to member mId. The Java method generated by EB^{3}TG for this action is represented in Fig. 8. The JDBC (Java Database 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. However, 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. 8.

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 (Fig. 4). 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.

4 CONCLUSION

We have presented an overview of EB^{3}TG, a tool for synthesizing Java programs that execute relational DB transactions that correspond to EB^3 attribute definitions. 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. By focusing on the translation of attribute definitions, the resulting transactions do not take the behaviour specified by the EB^3 process expression into account. This work must now be coupled with the analysis and/or the interpretation of EB^3 process expressions. This approach is radically different from paradigms widely used for specifying IS. The aim of the APIS project is to automate the synthesis of programs such that software engineers may focus on IS analysis and specification phases.

REFERENCES

Batanado, P. (2005). Synthèse des transactions de base de données relationnelle à partir de définitions d'attributs EB^3. Master's thesis, Département d'informatique, Université de Sherbrooke, Québec.

Bolognesi, T. and Brinksma, E. (1987). Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1).

Cousineau, G. and Mauny, M. (1998). *The functional approach to programming*. Cambridge University Press, Cambridge.

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

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

Elmasri, R. and Navathe, S. (2004). *Fundamentals of Database Systems*. Addison-Wesley, fourth edition.

Fraikin, B. and Frappier, M. (2002). EB3PAI: an interpreter for the EB^3 specification language. In *15th Intern. Conf. on Software and Systems Engineering and their Applications (ICSSEA 2002)*, Paris, France. CMSL.

Fraikin, B., Frappier, M., and Laleau, R. (2005). State-based versus event-based specifications for information systems: a comparison of B and EB^3. *Software and Systems Modeling*, 4(3):236–257.

Frappier, M., Fraikin, B., Laleau, R., and Richard, M. (2002). Apis - automatic production of information systems. In *AAAI Spring Symposium*, pages 17–24, Stanford, USA. AAAI Press.

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

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

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

Gervais, F., Frappier, M., and Laleau, R. (2005a). Generating relational database transactions from recursive functions defined on EB^3 traces. In *SEFM 2005 - 3rd IEEE International Conference on Software Engineering and Formal Methods*, Koblenz, Germany. IEEE Computer Society Press.

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

Hoare, C. (1985). *Communicating Sequential Processes*. Prentice-Hall.

Jackson, M. (1983). *System Development*. Prentice-Hall.

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

Mammar, A. (2002). *Un environnement formel pour le développement d'applications base de données*. PhD thesis, CNAM, Paris, France.

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

Prowell, S., Trammell, C., Linger, R., and Poore, J. (1999). *Cleanroom Software Engineering: Technology and Process*. Addison-Wesley.

Terrillon, J.-G. (2005). Description comportementale d'interfaces Web. Master's thesis, Département d'informatique, Université de Sherbrooke, Québec.

```
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));

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;
```

Figure 6: 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 recursive call
  Clues : The attribute recursive call 'member.loanDuration'
          must have exactly 2 parameters
```

Figure 7: Two examples of error messages.

```
public static void Transfer(int bId,int mId){
  try {
    connection.createStatement().executeUpdate(
        "CREATE TABLE eb3Tempmember ( "memberKey  numeric(5))");
    connection.createStatement().executeUpdate(
        "INSERT INTO  eb3Tempmember (memberKey) values("+mId+")");

    ResultSet rset0 =connection.createStatement().
      executeQuery("SELECT C.memberKey,A.nbLoans+1 "+
      "FROM eb3Tempmember C,member A "+
      "WHERE C.memberKey = "+mId+" "+
      "AND A.memberKey = C.mId ");

    ResultSet rset1 = connection.createStatement().
      executeQuery("SELECT G.borrower,E.nbLoans-1 "+
      "FROM loan G,member E "+
      "WHERE G.bookKey = "+bId+" "+
      "AND G.borrower NOT IN ( "+
      "SELECT C.memberKey "+
      "FROM eb3Tempmember C "+
      "WHERE C.memberKey = "+mId+" ) "+
      "AND E.memberKey = G.borrower ");

    ResultSet rset2 = connection.createStatement().
      executeQuery("SELECT D.loanDuration "+
      "FROM member D WHERE D.memberKey = "+mId+" ");
    String var0 = ((rset2.next())?rset2.getDouble(1)+"":"null");

    connection.createStatement().executeUpdate("UPDATE loan SET "+
      "borrower = "+mId+" WHERE bookKey = "+ bId +" ");

    connection.createStatement().executeUpdate("UPDATE loan SET "+
      "dueDate = SYSDATE+"+var0+" WHERE bookKey = "+ bId +" ");

    while(rset0.next()) {
      connection.createStatement().executeUpdate(
      "UPDATE member SET nbLoans = "+rset0.getDouble(2)+ " "+
      "WHERE memberKey = "+ rset0.getDouble(1));
    }

    while(rset1.next()) {
      connection.createStatement().executeUpdate(
      "UPDATE member SET nbLoans = "+rset1.getDouble(2)+ " "+
      "WHERE memberKey = "+ rset1.getDouble(1));
    }

    connection.createStatement().
      executeUpdate("DROP TABLE eb3Tempmember");
    connection.commit();
  } catch ( Exception e ) {
    try{
      connection.createStatement().
      executeUpdate("DROP TABLE   eb3Tempmember");
      connection.rollback();
    } catch (SQLException s){ System.err.println(s.getMessage());}
     System.err.println(e.getMessage());}
}
```

Figure 8: Java method for action Transfer.