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). EB³ (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

EB³ (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 EB³, 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 EB³ includes a process and a formal notation to described a precise and complete specification of the input-output behaviour of IS. An EB³ specification is composed of five parts:

1. A user requirements class diagram describes the entity types and associations of the IS, and their
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 EB$^3$ 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.

### 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 attribute value for a key value given as input, while a non-key attribute definition outputs the attribute value for a key value given as input. This diagram is based on entity-relationship (ER) model concepts (Elmasri and Navathe, 2004) and uses a UML-like graphical notation.

1. A process expression, denoted by main, defines the valid input traces of the system.
2. 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.
3. Recursive functions, defined on the valid input traces of main, assign values to entity type and associations attributes.
4. 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 $\sigma$ to trace $t$, and let $[\ ]$ denote the empty trace. The behaviour of the IS is defined as follows.

```plaintext
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.
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 ∈ T(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 \( \mathcal{F}(bK Set) \) 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 \( \text{input : expr} \), like Acquire(b1d,t1) : 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 \( \text{front}(s) \), 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), ...). For instance, we have the following values for attribute title:

\[
\begin{align*}
\text{title}([[]], b_1) & \overset{(I1)}{=} \bot \\
\text{title}([\text{Register}(m_1)], b_1) & \overset{(I5)}{=} \text{title}([[]], b_1) \overset{(I1)}{=} \bot \\
\text{title}([\text{Register}(m_1)], \text{Acquire}(b_1, t_1), b_1) & \overset{(I2)}{=} t_1
\end{align*}
\]

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 \( \text{input : expr} \) 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-
In this section, we introduce the main algorithm for synthesizing relational DB transactions that correspond to $\text{EB}^3$ attribute definitions and we present the $\text{EB}^3\text{TG}$ tool.

### 3.1 Main Algorithm

$\text{EB}^3$ 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 $\text{Att}(a)$ 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 $\text{EB}^3$ action $a$**
3. **analyse the input clauses**
4. **determine $\text{Att}(a)$**
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 $\text{EB}^3$ 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 $a(\overline{p}) : \text{expr}$ in its definition. To compute $T(a)$, a function $\text{table}$, generated in step (1), associates each attribute to its table in the DB. Hence, $T(a) = \{ \text{table}(b) \mid b \in \text{Att}(a) \}$.

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

Nevertheless, when expression \( \text{expr} \) in an input clause of the form \( \text{input} : \text{expr} \) contains \( \text{if then else} \) expressions, then we must analyse the different conditions in the \( \text{if} \) 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 \( \text{if} \) predicates; their construction and analysis are detailed in (Gervais et al., 2004). For instance, the \( \text{if} \) predicates in the conditional term of input clause \( \text{Transfer} \) in \( \text{nbLoans} \) determine two key values for \( mId : \text{mId}' \) and \( \text{borrower(bId)} \). Figure 4 shows the decision tree obtained for this input clause.

For the sake of concision, expression \( \text{front}(s) \) has been removed from the attribute recursive calls. The first leaf corresponds to condition \( \text{mId} = \text{mId}' \), and the second leaf to condition \( \text{mId} \neq \text{mId}' \land \text{mId} = \text{borrower(bId)} \). The last leaf is the recursive call of \( \text{nbLoans} \) 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 \( \text{expr} \) of the form \( \text{eKey(front(s))} - \{ k \} \) 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 \( \text{expr} \) of the form \( \text{eKey(front(s))} \cup S \), sets \( \text{eKey(front(s))} \) and \( S \) are not necessarily disjoint. Note that an expression \( f[(\text{front(s)})] \) 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 \( \text{Discard} \) is:

\[
\begin{align*}
\text{TRANSACTION} & \quad \text{Discard(mId : bK\_Set)} \\
\text{DELETE FROM} & \quad \text{book} /* delete statement */ \\
\text{WHERE} & \quad \text{bookKey} = \_\text{bId}; \\
\text{COMMIT}; & \\
\end{align*}
\]

Let us note that this transaction should be executed only when \( \text{Discard} \) 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 \( \text{Acquire} \) is:

\[
\begin{align*}
\text{TRANSACTION} & \quad \text{Acquire(bId : bK\_Set,bTitle : T)} \\
/* update statement */ \\
\text{UPDATE} & \quad \text{book SET} \quad \text{title} = \_\text{bTitle} \\
\text{WHERE} & \quad \text{bookKey} = \_\text{bId}; \\
/* test to determine whether the update has been successful */ \\
\text{IF} & \quad \text{SQL\%NotFound} \quad \text{THEN} \\
/* insert statement */ \\
\text{INSERT INTO} & \quad \text{book(bookKey,title)} \\
\text{VALUES} & \quad (\_\text{bId},\_\text{bTitle}); \\
\text{END}; \\
\text{COMMIT}; & \\
\end{align*}
\]

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

### 3.2 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 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
the sake of portability, we have chosen to use the lat-
example, Oracle is the chosen DBMS.

type and association of the system. Referential con-
stRAINTS are also automatically generated at the end to

debate mutual references between tables. For this

generated for the library management system is pre-
defined in Fig. 6. A table is created for each entity
type and association of the system. Referential con-
straintS 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 in-
stance, Fig. 7 shows two examples of syntax errors. In
the first example, keyword *match* is missing at col-
umn 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 defi-
tion. 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
method *Lend(bld, mld)* is to transfer the loan of book
*bld* to member *mld*. The Java method generated by
**EB**³**TG** for this action is represented in Fig. 8. The
**JDBC** (Java Database Connectivity) technology al-

dows Java programs to access the DBMS. Two classes
of the **JDBC** programming interface may execute
**SQL** statements to update and/or to query DB: *Prepare-
Statement* and *Statement*. The former is more ef-
cient 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 lat-
ter class. Method *createStatement()* creates a new
object of class *Statement*, while methods *execute-
Update(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 up-
dates. For instance, the analysis of attribute *nbLoans*
requires the construction of a decision tree (Fig. 4). In
lines 8 and 14, *rsel0* and *rsel1* 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
*bld*. 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 defini-
tions. Our programs introduce some overhead, be-
cause 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

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 Net-
works and ISDN Systems*, 14(1).

proach to programming*. Cambridge University Press,
Cambridge.


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.
1 bookStore.txt:40:9: expecting "with", found 'NULL'
2
3 >>Error in :
4 Attribute definition : loan.dueDate
5 Action : Lend(_,mId)
6 Cause : Invalid number of parameters in attribute recursive call
7 Clues : The attribute recursive call 'member.loanDuration'
8 must have exactly 2 parameters

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