How to Synthesize Relational Database Transactions From EB^3 Attribute Definitions?

F. Gervais[1,2], M. Frappier[2] and R. Laleau[3]

[1] Laboratoire CEDRIC, Institut d'Informatique d'Entreprise
18 Allée Jean Rostand, 91025 Évry Cedex, France
frederic.gervais@usherbrooke.ca

[2] GRIL, Département d'informatique, Université de Sherbrooke
Sherbrooke, Québec, Canada J1K 2R1
marc.frappier@usherbrooke.ca

[3] Laboratoire LACL, Université de 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). Attributes, linked to entities and associations of an IS, are computed in EB^3 by recursive functions on the valid traces of the system. In this paper, we show how to synthesize relational database transactions 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 key values affected by each action. To avoid problems with the sequencing of SQL statements in the transactions, temporary variables and/or tables are introduced for these key values.

1 Introduction

We are mainly interested in the formal specification of information systems (IS). In our viewpoint, an IS is a system that helps an organization to collect and to manipulate all its relevant data. The use of formal notation and techniques is justified for some systems when the data and/or their manipulation are considered as critic. The EB^3 [6] formal language has been specially created for that aim.

Example. The example used in this paper is a library management system. The system has to manage book loans to members. In particular, a member can transfer his loan to another member. A book can be lent by only one member at once. Figure 1 shows the user requirements class diagram of the example.

An Overview of EB^3. EB^3 is a trace-based formal specification language that can describe the input-output behaviour of an IS. The inputs are the events received by the system, like action `Lend` in the example. The outputs are computations on attribute values in answer to an input event, *e.g.*, a function that returns the number of loans of

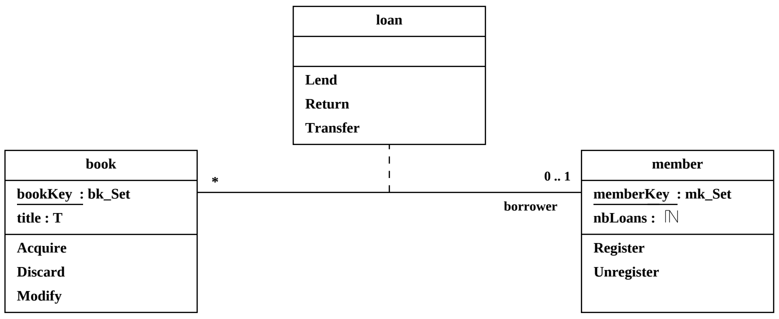

Fig. 1. User requirements class diagram of the library

a member. An EB^3 specification consists of the following elements: i) a user requirements class diagram which includes entities, associations, and their respective actions and attributes; ii) a process expression, denoted by `main`, which defines the valid input event traces; iii) recursive functions, defined on the traces of `main`, that assign values to entity and association attributes; iv) input-output rules, which assign an output to each valid input event trace. Indeed, 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 the sequence of the valid input events accepted so far in the execution of the system, let $\texttt{trace::}\sigma$ denote the right append of element σ to trace `trace`, and let `[ ]` denote the empty trace. Then, we have:

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

The EB^3 notation for process expressions is similar to Hoare's CSP [9]. The complete syntax and semantics of EB^3 can be found in [6] and the process expression for the example in [8]. EB^3 expressions are close to the user view and complex constraints inside and between entities are easy to specify in EB^3 [4]. The input-output rules of the example are described in [7].

Outline. The APIS project [5] aims at generating IS from EB^3 specifications. There already exists an interpreter, called EB3PAI [3], for EB^3 process expressions. It allows one to generate IS from correct EB^3 specifications. Nevertheless, the computation of attribute values in EB3PAI is not taken into account yet, and transactions are considered as black boxes. In this paper, we focus on the synthesis of relational database transactions that correspond to EB^3 attribute definitions. Thus, we will be able to efficiently interprete EB^3 specifications for the purpose of software prototyping and requirements validation. The synthesized imperative programs are of the same algorithmic complexity as those manually generated by a programmer. Hence, they could also be used in concrete implementations of EB^3 specifications. EB^3 attribute definitions are presented in Sect. 2. In Sect. 3, we show how to generate SQL statements that correspond to EB^3 attribute definitions. Finally, Sect. 4 concludes the paper with some comments and perspectives.

2 EB^3 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 pattern-matching-like style, as in CAML [1]. 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. A key definition outputs a set of key values, while a non-key attribute definition outputs the attribute value for a key value given as an input parameter. For instance, the key of entity type `book` is defined by function $bookKey$ in Fig. 2. $bookKey$ has a unique input parameter $s \in \mathcal{T}(\mathsf{main})$, *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 attribute $title$ is defined in Fig. 2.

$bookKey(s : \mathcal{T}(\mathsf{main})) : \mathbb{F}(bk\_Set) \quad \triangleq$	$title(s : \mathcal{T}(\mathsf{main}), bId : bk\_Set) : T \quad \triangleq$	
match $last(s)$ **with**	**match** $last(s)$ **with**	
$\perp : \emptyset,$	$\perp : \perp,$	(I1)
$\mathsf{Acquire}(bId, \_) : bookKey(front(s)) \cup \{bId\},$	$\mathsf{Acquire}(bId, ttl) : ttl,$	(I2)
$\mathsf{Discard}(mId) : bookKey(front(s)) - \{bId\},$	$\mathsf{Discard}(bId) : \perp,$	(I3)
$\_ : bookKey(front(s));$	$\mathsf{Modify}(bId, ttl) : ttl,$	(I4)
	$\_ : title(front(s), bId);$	(I5)

Fig. 2. Examples of EB^3 attribute definitions

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 $last(s)$, then the corresponding expression $expr$ is computed; otherwise, the function is recursively called with the first elements of s except the last one, denoted by $front(s)$. This case corresponds to the last input clause with symbol '_'. EB^3 attribute definitions always include $\perp$, 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 $title$:

$$\begin{array}{l}
title([\,], b_1) \overset{(I1)}{=} \bot \\
title([\mathsf{Register}(m_1)], b_1) \overset{(I5)}{=} title([\,], b_1) \overset{(I1)}{=} \bot \\
title([\mathsf{Acquire}(b_1, t_1)], b_1) \overset{(I2)}{=} t_1 \\
title([\mathsf{Acquire}(b_1, t_1), \mathsf{Register}(m_1), \mathsf{Modify}(b_1, t_2)], b_1) \overset{(I3)}{=} t_2
\end{array}$$

In the first example, the value is obtained from input clause (I1), since $last([\,]) = \bot$. In the second example, we first applied the wild card clause (I5), since no input clause matches Register, and then (I1). In the last examples, the value is obtained directly from (I2) and (I3), respectively.

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 to determine the key values affected by an action. For instance, the input clause for Transfer in attribute $nbLoans$ is:

$$\begin{array}{ll}
\mathsf{Transfer}(bId, mId') : & \textbf{if } mId = mId' \textbf{ then } nbLoans(front(s), mId) + 1 \\
& \textbf{else if } mId = borrower(front(s), bId) \textbf{ then} \\
& \quad nbLoans(front(s), mId) - 1 \textbf{ end end},
\end{array}$$

The key of $nbLoans$ is mId, and the **if** predicates determine two key values for mId: mId' and $borrower(front(s), bId)$.

3 Synthesizing Relational Database Transactions

In the EB^3 semantics, when a new event of action a is accepted by process expression `main`, then all the attributes affected by a must be updated. To generate a RDBMS transaction for each EB^3 action a, we must analyse 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. The general algorithm is the following:

```
for each action $a$ of the EB$^3$ specification
    analyse the input clauses of EB$^3$ attribute definitions
    determine the tables $T(a)$ affected by $a$
    for each $t$ in $T(a)$
        determine the key values to delete
        determine the key values to insert and/or to update
    define the transaction for $a$
```

In the remainder of this paper, the SQL 92 norm [10] is used for SQL queries, while a procedural pseudo-language is used for transactions.

Definition of Temporary Variables and Temporary Tables. The analysis of the input clauses is summed up in this paper; the algorithms are presented in [7]. When a pattern matching condition evaluates to true, an assignment of a value for each free variable in the input clause has been determined. 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 bounded by the pattern matching. We use a binary trees called *decision trees* to analyse the **if** predicates; their construction and analysis are detailed in [8].

When key values are determined from predicates involving 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. A temporary variable is defined when a unique key value is determined, while a temporary table is used to characterize several key values. For instance, if we need the collection of books lent by member mId, then the following table is defined:

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

We do not use views, because we want to consider the values of the data before any modification. Thus, the content of the temporary tables is evaluated only once, at the beginning of the transaction. 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 [7].

Definition of Transactions. For defining transactions, all the SQL statements are grouped by table. Thanks to the analysis of the input clauses, the key values to delete are distinguished from the other key values. The **DELETE** statements are grouped at the beginning of each table's list of instructions. For instance, the transaction generated for action Discard is:

```
TRANSACTION Discard(mId : BOOKID)
    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 trasaction generated for Acquire is:

```
TRANSACTION Acquire(bId : BOOKID,bTitle : T)
    VAR R : ResultSet /* define a temporary variable for the test */
        SELECT bookKey INTO R /* extract bId from book */
        FROM book
        WHERE bookKey = #bId;
    IF R is not empty /* test to determine whether bId is in book */
    THEN UPDATE book SET title = #ttl /* update statement */
            WHERE bookKey = #bId;
```

```
ELSE  INSERT INTO book(bookKey,title) /* insert statement */
      VALUES (#bId,#ttl);
END;
COMMIT;
```

Such a transaction could be simplified by the analysis of EB^3 process expressions.

4 Conclusion

In this paper, we have presented an overview of an algorithm that synthesizes relational database 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. Our programs introduce some overhead, because they systematically store the current values of attributes before updating the database, in order to ensure correctness. We plan to optimize these programs by analysing dependencies between update statements and avoid, when possible, these intermediate steps. 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. Several papers deal with the synthesis of relational implementations. Most of the time, refinement techniques are used, like in [2] for Z and [11] for B specifications, which are orthogonal in specification style to EB^3 [4].

References

1. Cousineau, G., Mauny, M.: The Functional Approach to Programming. Cambridge University Press, Cambridge (1998)
2. Edmond, D.: Refining Database Systems. In Proc. ZUM'95, Limerick, Ireland, 7-9 September 1995. LNCS, Vol. 967, Springer-Verlag (1995) 25-44
3. Fraikin, B., Frappier, M.: EB3PAI: an Interpreter for the EB^3 Specification Language. In Proc. 15th Intern. Conf. on Software and Systems Engineering and their Applications, Paris, France, 3-5 December 2002. CMSL, Paris (2002)
4. Fraikin, B., Frappier, M., Laleau, R.: State-Based versus Event-Based Specifications for Information Systems: a Comparison of B and EB^3. Software and System Modeling, to appear
5. Frappier, M., Fraikin, B., Laleau, R., Richard, M.: APIS - Automatic Production of Information Systems. In Proc. AAAI Spring Symposium, Stanford, USA, 25-27 March 2002. Techn. Rep. SS-02-05, AAAI Press (2002) 17-24
6. Frappier, M., St-Denis, R.: EB^3: an Entity-Based Black-Box Specification Method for Information Systems. Software and System Modeling, **2**(2) (2003) 134-149
7. Gervais, F., Frappier, M., Laleau, R.: EB^3 Attribute Definitions: Formal Language and Application. Technical Report 700, CEDRIC, Paris, France (2005)
8. Gervais, F., Frappier, M., Laleau, R.: Synthesizing B Substitutions for EB^3 Attribute Definitions. Technical Report 683, CEDRIC, Paris, France (2004)
9. Hoare, C. A. R.: Communicating Sequential Processes. Prentice-Hall (1985)
10. ISO: Database Language SQL. International Standard ISO/IEC JTC1/SC21, doc. 9075 N5739 (1992)
11. Mammar, A.: Un environnement formel pour le développement d'applications base de données. Ph.D. thesis, CNAM, Paris (2002)