CEDRIC

Technical Report 683:

Synthesizing B Substitutions for EB^3 Attribute Definitions

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

First version in November 2004
Current version revised in January 2005

[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`

[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). Attributes, linked to entities and associations of an IS, are computed in EB^3 by recursive functions on the valid traces of the system. On the other hand, B is a state-based formal language also well adapted for the specification of IS. In this technical report, we show how to synthesize B substitutions from EB^3 attribute definitions. The static part of the B specification is generated from a class diagram. Each action in the EB^3 specification is translated into a B operation. The substitutions are obtained by an analysis of CAML-like patterns used in the recursive functions that define the attributes in EB^3. We present two ways to generate B substitutions. Either we use general formulas, called substitution formulas, that characterize all the B substitutions associated to an attribute and then we synthesize usual B substitutions from these formulas, or we generate B substitutions directly. Our techniques are illustrated by an example of a library management system.

Contents

Chapter 1

Introduction

There exist several paradigms to specify information systems. Broadly speaking, an information system is a system that helps an organization to collect and to manipulate all its relevant data. We focus on the formal specification of such systems. We are mainly interested by two approaches dedicated to the formal specification of information system: UML-B [Ngu98] and EB^3 [FSD03]. The former provides techniques to generate B formal specifications from UML semi-formal class diagrams, while the latter is a trace-based formal language specially created for the specification of information systems.

We aim at defining an integrated approach that combines both languages in order to consider not only more viewpoints, but also more verification purposes. In that goal, EB^3 provides process expressions that represent the valid traces of the system and recursive functions that computes attribute values from the valid traces. This view is radically different from the B model of an information system. For instance, attributes are represented by state variables and actions are specified with operations. In EB^3, one must analyse all the traces of the system to compute new attribute values, while in B, each operation specifies which and how state variables are changed by its execution.

In this report, we focus on the translation of EB^3 attribute definitions into B. We aim at synthesizing the corresponding substitutions. This is an important step towards the generation of a complete B specification from an EB^3 specification. This translation from EB^3 to B will enable the proof of static properties for EB^3 specification by proving them on their corresponding B specification. This work is illustrated by an example of a library management system. The paper is organized as follows. Chapter 2 is a general introduction to EB^3. Chapter 3 deals with EB^3 attribute definitions. Chapter 4 is a short introduction to the interests of combining EB^3 and B and to the generation of the static part of the B specification. Then, Chapter 5 shows how to synthesize B substitutions for EB^3 attribute definitions by using substitution formulas, while Chapter 6 shows how to directly synthesize such substitutions. Finally, Chapter 7 concludes this technical report with some remarks and perspectives.

Chapter 2

Specifying Information Systems with the EB^3 Method

2.1 An Overview of EB^3

The core of EB^3 [FSD03] includes a specification process and a formal notation for progressing from use cases to a complete and precise specification of input-output behaviour. An EB^3 specification consists of the following elements:

1. a user requirements class diagram which includes entities, associations, and their respective actions and attributes. These diagrams are based on entity-relationship model concepts [Elm04].
2. a process expression, denoted by `main`, which defines valid input traces;
3. recursive functions, defined on the traces of `main`, that assign values to entity and association attributes;
4. input-output rules, which assign an output to each valid input trace.

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

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

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

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 $\mathtt{a}(t_1, ..., t_n)$ 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 (as we shall see below); 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.

2.2 The Process to Obtain an EB^3 Specification

We illustrate the main aspects of an EB^3 specification using a simple library management system whose requirements are stated below. Definition of attributes with recursive functions will be discussed in Chapter 3.

The User Requirements of a Library Management System

In this section, we provide an informal description of the user requirements of a library system. The library system has to manage book loans to members and was first presented in [FFL03].

1. A book is acquired by the library. It can be discarded, but only if it is not lent.
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.

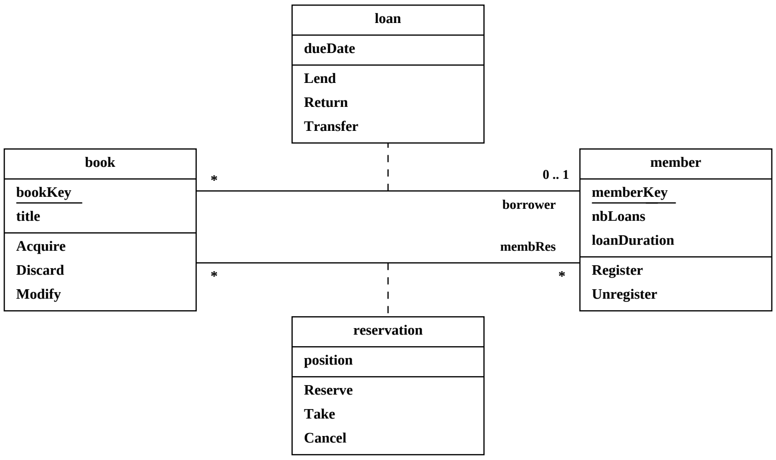

Figure 2.1: EB^3 specification: user requirements class diagram of the library IS

5. A member can reserve a book if it is lent by another member.
6. A book can be lent by only one member at once.
7. A book can be reserved, even by several members, but only if it is lent.
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.

The User Requirements Class Diagram

Figure 2.1 shows the user requirements class 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 2.2 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`.

The Process Expressions of Entities and Associations

In [FSD03], Frappier and St Denis describe a strategy to design an EB^3 specification, in which each entity type and association is represented by a process expression. The interactions between entities are naturally expressed by composing entities in parallel using operator `||` in the process `main` (see Fig. 2.3).

Simple ordering constraints (basic scenarios) can be expressed easily using the process algebra operators. To do so, we can take each entity type from the requirements class diagram (book and member) and express its ordering constraints on input events by using a process expression (see Fig. 2.4).

```
Acquire(bId:bookKey_Set,bTitle:Title_Type):void
Discard(bId:bookKey_Set):void
Modify(bId:bookKey_Set,newTitle:Title_Type):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 2.2: EB^3 specification: signature of actions

```
main =
   ( ||| bId : bookKey_Set : book(bId)^* )
||
   ( ||| mId : memberKey_Set : member(mId)^* )
```

Figure 2.3: EB^3 specification: process `main`

```
book(bId : bookKey_Set) =
 Acquire(bId,_).
 (
     ( | mId : memberKey_Set : loan(mId,bId)^* )
  ||
     loanCycleBook(bId)^*
  ||
     ( ||| mId : memberKey_Set : reservation(mId,bId)^* )
  ||
     Modify(bId,_)^*
 ).
 Discard(bId)

member(mId : memberKey_Set) =
 Register(mId,_).
 (
     ( ||| bId : bookKey_Set : loan(mId,bId)^* )
  ||
     ( ||| bId : bookKey_Set : reservation(mId,bId)^* )
 ).
 Unregister(mId)
```

Figure 2.4: EB^3 specification: entity process definitions

```
loan(mId : memberKey_Set, bId : bookKey_Set) =
      loanNominal(mId,bId)
  ||
      loanController(mId,bId)^*

loanNominal(mId : memberKey_Set, bId : bookKey_Set) =
  ( Lend(bId,mId,_) | Take(bId,mId,_) | Transfer(bid,mId,_) ).
  ( Return(bId) | lambda )

loanController(mId : memberKey_Set, bId : bookKey_Set) =
      (     nbLoans(trace,mId) < maxNbLoan
        and isNotReserved(trace,bId)
      ) ==> Lend(bId,mId)
    |
      nbLoans(trace, mId) < maxNbLoan ==> Take(bId,mId)
    |
      borrowed(trace,bId) ==>
      ( | mId : memberKey_Set :
              ( mId != currentBorrower(bId) ==> Transfer(bId,mId) )
      )
```

Figure 2.5: EB^3 Specification: `loan` process definition

The entity processes in turn call the processes `loan` and `reservation` which describe the behaviour of the corresponding associations (see Fig. 2.5 and 2.6). The multiplicity of an association (*e.g.,* 1..*,*, 0..1, and so on) is expressed in the process expressions of the entities by selecting an appropriate quantification operator to encapsulate the call to the association process expression. Several patterns have been defined to translate requirements class diagrams into process expressions (see [FSD03]).

Associations `loan` and `reservation` are specified in the same way. For instance, in Fig. 2.5, a loan is specified with two processes in parallel, `loanNominal` and `loanController`.

The first process is an instantiation of the **producer-modifier-consumer pattern** [FSD03], without any modifier (one could have defined an action `Renew` for instance). Action `Transfer` could have been considered as a consumer too, but in this example, `Transfer` is defined as a producer and, consequently, special symbol `lambda` is used to represent an internal action as a consumer. So parameter mId in `Transfer` is the member that transfers the book bId. To ensure that a lent book is eventually returned, an auxiliary process that controls association `loan` for each book in Fig. 2.4 is defined in Fig. 2.7.

The second process in `loan` is another process controller that specifies when an action in process `loanNominal` can be executed. Actions are thus guarded (by the predicates before symbol `==>`). Guards are computed thanks to recursive functions on the finite traces accepted by process expression `main`. Definition of entity and association attributes, like `nbLoans`, also uses recursive functions; this is the topic of Chapter 3. The other recursive functions used in this example are specified in [FFL03].

```
reservation(mId : memberKey_Set, bId : bookKey_Set) =
      reservationNominal(mId,bId)
  ||
      reservationController(mId,bId)^*

reservationNominal(mId : memberKey_Set, bId : bookKey_Set) =
  Reserve(bId,mId).
  ( Take(bId,mId,_) | Cancel(bId,mId) )

reservationController(mId : MEMBERID, bId : BOOKID) =
      canBeReserved(trace,mId,bId) ==> Reserve(bId,mId)
    |
      isFirst(trace,mId,bId) ==> Take(bId,mId,_)
```

Figure 2.6: EB^3 Specification: `reservation` process definition

```
loanCycleBook(bId : bookKey_Set) =
  Lend(bId,_,_) .
  Transfer(bid,_,_)^* .
  Return(bId)
```

Figure 2.7: EB^3 Specification: `loanCycleBook` process controller

Chapter 3

EB^3 Attribute Definitions

The main steps to obtain an EB^3 specification have been presented in Sect. 2.2. In this chapter, we deal with attribute definitions.

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

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 without quantifiers 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, $var(t) = \emptyset$.

3.1 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. 2.1) is

$$bookKey \;:\; \mathcal{T}(\mathsf{main}) \to \mathbb{F}(bookKey\_Set) \;,$$

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

$$\begin{array}{l} eKey\ (s : \mathcal{T}(\mathsf{main})) : \mathbb{F}(T_1 \times \cdots \times T_m) \ \triangleq \\ \quad \textbf{match}\ last(s)\ \textbf{with} \\ \qquad \begin{array}{lcl} \perp & : & u_0, \\ a_1(\overrightarrow{p_1}) & : & u_1, \\ \ldots & & \\ a_n(\overrightarrow{p_n}) & : & u_n, \\ \_ & : & eKey(front(s)); \end{array} \end{array} \tag{3.1}$$

Expressions $\perp : u_0$, $a_1(\overrightarrow{p_1}) : u_1$, ..., $a_n(\overrightarrow{p_n}) : u_n$ and $\_ \ : \ eKey(front(s))$ are called *input clauses*. Expression $last(s)$ returns $\perp$ 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 EB^3, 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 $k_1^j, \ldots, k_{m_j}^j$ the key attributes of entity type e_j. The key of ass is then defined by considering all the key attributes of entity types $e_1, \ldots, e_l$:

$$ass \;:\; \mathcal{T}(\mathsf{main}) \to \mathbb{F}(T_{k_1^1} \times \cdots \times T_{k_{m_1}^1} \times \cdots \times T_{k_1^l} \times \cdots \times T_{k_{m_l}^l}) \;,$$

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

$$loan \;:\; \mathcal{T}(\mathsf{main}) \to \mathbb{F}(bookKey\_Set \times memberKey\_Set) \;,$$

3.2 Non-Key Attributes

In EB^3, each non-key attribute b_i of an entity type or an association is defined by a function b_i of type

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

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, EB^3 recursive functions are total. For instance, the type of the function associated to attribute *borrower* is

$$borrower \ : \ \mathcal{T}(\mathsf{main}) \times bookKey\_Set \rightarrow memberKey\_Set \ .$$

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

$$\begin{array}{l} b_i \ (s : \mathcal{T}(\mathsf{main}), \overrightarrow{k} : T_1 \times \cdots \times T_m) : T_i \ \triangleq \\ \quad \textbf{match } last(s) \textbf{ with} \\ \qquad \begin{array}{lcl} \perp & : & u_0, \\ a_1(\overrightarrow{p_1}) & : & u_1, \\ \ldots & & \\ a_n(\overrightarrow{p_n}) & : & u_n, \\ \_ & : & b(front(s), \overrightarrow{k}); \end{array} \end{array} \tag{3.2}$$

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 requirements class 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.

3.3 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 paper:

$$\begin{array}{l} b\ (\overrightarrow{k}) : T \ \triangleq \\ \quad \begin{array}{lll} \perp & : & u_0, \\ a_1(\overrightarrow{p_1}) & : & u_1, \\ \ldots & & \\ a_n(\overrightarrow{p_n}) & : & u_n; \end{array} \end{array} \tag{3.3}$$

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.

3.4 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

$$\exists\, (var(\overrightarrow{p_i}) - \overrightarrow{k}) \ \bullet \ last(s) = a_i(\overrightarrow{p_i})\ \rho \ . \tag{3.4}$$

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:

$$\begin{array}{l} b\ (\overrightarrow{k}) : T \ \triangleq \\ \quad \begin{array}{lll} \ldots & & \\ a(\overrightarrow{p_1}) & : & u_1, \\ \ldots & & \\ a(\overrightarrow{p_n}) & : & u_n; \\ \ldots & & \end{array} \end{array} \tag{3.5}$$

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 (3.4) 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 *borrower* is:

$$\begin{array}{llll}
\multicolumn{4}{l}{borrower(s : \mathcal{T}(\mathsf{main}), bId : bookKey_Set) : memberKey_Set \triangleq} \\
\multicolumn{4}{l}{\quad \textbf{match}\ last(s)\ \textbf{with}} \\
\quad\quad \perp & : & \perp, & \text{(I1)} \\
\quad\quad \mathsf{Lend}(bId, mId, _) & : & mId, & \text{(I2)} \\
\quad\quad \mathsf{Return}(bId) & : & \perp, & \text{(I3)} \\
\quad\quad \mathsf{Transfer}(bId, mId, _) & : & mId, & \text{(I4)} \\
\quad\quad \mathsf{Take}(bId, mId, _) & : & mId, & \text{(I5)} \\
\quad\quad _ & : & borrower(front(s), bId); & \text{(I6)}
\end{array}$$

With the default input clauses (*e.g.,* (I1) and (I6)), 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 *borrower*.

$$\begin{array}{l}
borrower([\,], c_1) \stackrel{\text{(I1)}}{=} \perp \\
borrower([\mathsf{Acquire}(c_1, _)], c_1) \stackrel{\text{(I6)}}{=} borrower([\,], c_1) \stackrel{\text{(I1)}}{=} \perp \\
borrower([\mathsf{Acquire}(c_1, _), \mathsf{Register}(d_1, 21), \mathsf{Lend}(c_1, d_1, Classic)], c_1) \stackrel{\text{(I2)}}{=} d_1
\end{array}$$

In the first example, the value is obtained from input clause (I1), since $last([\,]) = \perp$. In the second example, we first applied the wild card clause (I6), since no input clause matches Acquire, and then (I1). In the third example, the value is obtained immediately from (I2) with the following substitution:

$$\theta = \{bId := c_1, mId := d_1\}$$

3.5 Properties

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 EB^3 specifications of attribute definitions satisfy the following consistency condition: when a non-key attribute b returns a value other than $\perp$ for a key value, then the key function should contain that key.

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

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.

3.6 An Example of EB^3 Attribute Definitions

Here are the attribute definitions for the small library system example presented in Sect. 2.2.

Entity Type Book

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

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

$borrower(bId) : memberKey\_Set \triangleq$
$\quad \mathsf{Lend}(bId, mId, \_) \ : \ mId,$
$\quad \mathsf{Return}(bId) \ : \ \perp,$
$\quad \mathsf{Transfer}(bId, mId, \_) \ : \ mId,$
$\quad \mathsf{Take}(bId, mId, \_) \ : \ mId;$

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

Entity Type Member

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

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

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

Association loan

$loan() : \mathbb{F}(bookKey\_Set \times memberKey\_Set) \triangleq$
$\quad \mathsf{Lend}(bId, mId, \_) \ : \ loan() \cup \{(bId, mId)\},$
$\quad \mathsf{Return}(bId) \ : \ loan() - \{(bId, borrower(bId))\},$
$\quad \mathsf{Transfer}(bId, mId, \_) \ : \ loan() - \{(bId, borrower(bId)\} \cup \{(bId, mId)\},$
$\quad \mathsf{Take}(bId, mId, \_) \ : \ loan() \cup \{(bId, mId)\};$

$dueDate(bId, mId) : DATE \triangleq$
$\quad \mathsf{Lend}(bId, mId, Permanent) \ : \ CurrentDate + 365,$
$\quad \mathsf{Lend}(bId, mId, Classic) \ : \ CurrentDate + loanDuration(mId),$
$\quad \mathsf{Return}(bId) \ : \ \perp,$

$$
\begin{array}{ll}
\quad \mathsf{Transfer}(bId, mId', type) : & \textbf{if } mId = mId' \\
& \textbf{then if } type = Permanent \\
& \qquad \textbf{then } CurrentDate + 365 \\
& \qquad \textbf{else } CurrentDate + loanDuration(mId) \\
& \qquad \textbf{end} \\
& \textbf{else if } mId = borrower(bId) \\
& \qquad \textbf{then } \bot \\
& \qquad \textbf{end} \\
& \textbf{end}, \\
\quad \mathsf{Take}(bId, mId, Permanent) : & CurrentDate + 365, \\
\quad \mathsf{Take}(bId, mId, Classic) : & CurrentDate + loanDuration(mId);
\end{array}
$$

Association reservation

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

$$
\begin{array}{ll}
\multicolumn{2}{l}{position(bId, mId) : \mathbb{N} \triangleq} \\
\quad \mathsf{Reserve}(bId, mId) : & card(membRes(bId)) + 1, \\
\quad \mathsf{Cancel}(bId, mId') : & \textbf{if } mId = mId' \\
& \textbf{then } \bot \\
& \textbf{else if } mId \in membRes(bId) \wedge \\
& \qquad\qquad position(bId, mId') < position(bId, mId) \\
& \qquad \textbf{then } position(bId, mId) - 1 \\
& \qquad \textbf{end} \\
& \textbf{end}, \\
\quad \mathsf{Take}(bId, mId', \_) : & \textbf{if } mId = mId' \\
& \textbf{then } \bot \\
& \textbf{else if } mId \in membRes(bId) \\
& \qquad \textbf{then } position(bId, mId) - 1 \\
& \qquad \textbf{end} \\
& \textbf{end};
\end{array}
$$

Let us recall that the above-mentionned 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`.

Chapter 4

Motivation and Goals

4.1 Two Different Ways to Represent Attributes

The main difference between EB^3 and B is that EB^3 is an event-based language, while B is a state-based language. The consequences on the attribute definitions are the following ones. In EB^3, the values of an attribute are defined from a recursive function on the valid traces of the system. For each attribute, its associated recursive function specifies what is the effect of each action of the system on the attribute values when this action is accepted and effectively executed by the system. In B, the specification is radically orthogonal. Attributes are defined as state variables of the system. Each B operation, that corresponds to an EB^3 action, specifies what are the substitutions on the state variables of the system, when this operation is executed.

Attribute definitions of EB^3 specifications describe how the values of the attributes evolve. The following is a general, high level abstract algorithm that computes the attribute values of the system:

initialize attribute values
$s := [\,]$
while true do
    receive valid input σ
    determine attributes b that are affected by the reception of σ
    for each attribute b
        determine key values $\overrightarrow{k}$ that are affected for b by the reception of σ
        for each $\overrightarrow{k}$
            compute the value of $b(s \leftarrow \sigma, \overrightarrow{k})$
    $s := s \leftarrow \sigma$

We aim at synthesizing B specifications that correspond to EB^3 attribute definitions. Since the attribute values of entity types and associations, that are represented in B by state variables, are modified by these recursive functions, the effects of attribute definitions correspond to substitutions in B. That is why, in our B translation of the system, the operations correspond to the actions of the EB^3 specification. Thus, we can generate from EB^3 attribute definitions the substitutions for the initialization and the operations of the B specification of the system.

Let us note that the precondition of an operation not only depends on the type of its input parameters, but also on the valid traces of the system described by EB^3 process expressions. In this paper, we only focus on the synthesis of B specifications from attribute definitions. The generation of preconditions of operations and invariant properties from the valid traces of the system is a work in progress. Consequently, preconditions described in the following operations consist only of typing constraints.

The algorithm for synthesizing B specifications from EB^3 attribute definitions consists of four steps:

1. generate the static part of the B machine from the user requirements class diagram and complete the missing types for attributes with the analysis of the signature of EB^3 attribute definitions,
2. generate the signature and precondition of operations from the signature of actions in EB^3,
3. generate substitutions for the initialization,
4. generate substitutions for the operations.

The first three steps are shown in this chapter, while the last one is discussed in the next two chapters.

4.2 Example in B

The following B specification can be generated from the EB^3 specification of the library management system described in Sect. 2.2.

MACHINE $B\_Library$

SEES ... `/* B machine defining operations on DATE */`

SETS
$memberKey\_Set$; $bookKey\_Set$; $Title\_Type$;
$Loan\_Type = \{Permanent, Classic\}$

VARIABLES
$memberKey$, $nbLoans$, $loanDuration$, $bookKey$, $title$, $loan$,
$dueDate$, $reservation$, $position$

INVARIANT
$memberKey \subseteq memberKey\_Set \ \wedge \ nbLoans \in memberKey \rightarrow NAT \ \wedge$
$loanDuration \in memberKey \rightarrow NAT \ \wedge \ bookKey \subseteq bookKey\_Set \ \wedge$
$title \in bookKey \rightarrow Title\_Type \ \wedge \ loan \in bookKey \nrightarrow memberKey \ \wedge$
$dueDate \in loan \rightarrow DATE \ \wedge \ reservation \in bookKey \leftrightarrow memberKey \ \wedge$
$position \in reservation \rightarrow NAT$

DEFINITION
$borrower(x) \triangleq loan(x)$;
$membRes(x) \triangleq reservation(x)$

INITIALISATION
$memberKey, nbLoans, loanDuration, bookKey, title, loan, dueDate,$
$reservation, position := \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset$

OPERATIONS
Acquire$(bId, bTitle) \triangleq$
PRE $bId \in bookKey\_Set \ \wedge \ bTitle \in Title\_Type$
THEN
$bookKey := bookKey \cup \{bId\} \ ||$
$title := title \cup \{bId \mapsto bTitle\}$
END;

Discard$(bId) \triangleq$
PRE $bId \in bookKey\_Set$
THEN
$bookKey := bookKey - \{bId\} \ ||$
$title := \{bId\} \lhd\!\!\!- title$
END;

Modify$(bId, newTitle) \triangleq$
PRE $bId \in bookKey\_Set \ \wedge \ newTitle \in Title\_Type$
THEN
$title := title \lhd\!\!\!+ \{bId \mapsto newTitle\}$
END;

Register$(mId, lD) \triangleq$
PRE $mId \in memberKey\_Set \ \wedge \ lD \in \mathbb{N}$
THEN
$memberKey := memberKey \cup \{mId\} \ ||$
$nbLoans := nbLoans \lhd\!\!\!+ \{mId \mapsto 0\} \ ||$
$loanDuration := loanDuration \lhd\!\!\!+ \{mId \mapsto lD\}$
END;

Unregister$(mId) \triangleq$
PRE $mId \in memberKey\_Set$
THEN
$memberKey := memberKey - \{mId\} \ ||$
$nbLoans := \{mId\} \lhd\!\!\!- nbLoans$
$loanDuration := \{mId\} \lhd\!\!\!- loanDuration$
END;

Lend$(bId, mId, typeOfLoan) \triangleq$
PRE $bId \in bookKey\_Set \ \wedge \ mId \in memberKey\_Set \ \wedge \ typeOfLoan \in Loan\_Type$
THEN
$loan := loan \cup \{(bId, mId)\} \ ||$
$nbLoans := nbLoans \lhd\!\!\!+ \{mId \mapsto 1 + nbLoans(mId)\} \ ||$
IF $typeOfLoan = Permanent$
THEN $dueDate := dueDate \lhd\!\!\!+ \{(bId, mId) \mapsto CurrentDate + 365\}$

ELSE $dueDate := dueDate \lhd\!\!- \{(bId, mId) \mapsto CurrentDate + loanDuration(mId)\}$
END
END;

Return$(bId) \triangleq$
PRE $bId \in bookKey\_Set$
THEN
$loan := loan - \{(bId, borrower(bId))\}$ ||
$nbLoans := nbLoans \lhd\!\!- \{borrower(bId) \mapsto nbLoans(borrower(bId)) - 1\}$ ||
$dueDate := \{(bId, mId)\} \lhd\!\!\!- dueDate$
END;

Transfer$(bId, mId, typeOfLOan) \triangleq$
PRE $bId \in bookKey\_Set \ \wedge \ mId \in memberKey\_Set \ \wedge \ typeOfLoan \in Loan\_Type$
THEN
$loan := loan - \{(bId, borrower(bId))\} \cup \{(bId, mId)\}$ ||
$nbLoans := nbLoans \lhd\!\!- \{(mId \mapsto nbLoans(mId) + 1),$
$(borrower(bId) \mapsto nbLoans(borrower(bId)) - 1)\}$ ||
IF $typeOfLoan = Permanent$
THEN $dueDate := (\{(bId, borrower(bId))\} \lhd\!\!\!- dueDate)$
$\lhd\!\!- \{(bId, mId) \mapsto CurrentDate + 365\}$
ELSE $dueDate := (\{(bId, borrower(bId))\} \lhd\!\!\!- dueDate)$
$\lhd\!\!- \{(bId, mId) \mapsto CurrentDate + loanDuration(mId)\}$
END
END;

Reserve$(bId, mId) \triangleq$
PRE $bId \in bookKey\_Set \ \wedge \ mId \in memberKey\_Set$
THEN
$reservation := reservation \cup \{(bId, mId)\}$ ||
$position := position \lhd\!\!- \{(bId, mId) \mapsto card(membRes(bId)) + 1\}$
END;

Take$(bId, mId, typeOfLoan) \triangleq$
PRE $bId \in bookKey\_Set \ \wedge \ mId \in memberKey\_Set \ \wedge \ typeOfLoan \in Loan\_Type$
THEN
$loan := loan \cup \{(bId, mId)\}$ ||
$nbLoans := nbLoans \lhd\!\!- \{mId \mapsto nbLoans(mId) + 1\}$ ||
IF $typeOfLoan = Permanent$
THEN $dueDate := dueDate \lhd\!\!- \{(bId, mId) \mapsto CurrentDate + 365\}$
ELSE $dueDate := dueDate \lhd\!\!- \{(bId, mId) \mapsto CurrentDate + loanDuration(mId)\}$
END ||
$reservation := reservation - \{(bId, mId)\}$ ||
$position := position \lhd\!\!- \{(x_1, x_2, y) \mid x_1 = bId \wedge y = position(bId, x_2) - 1 \wedge$
$x_2 \neq mId \wedge x_2 \in membRes(bId)\}$
END;

Cancel$(bId, mId) \triangleq$
PRE $bId \in bookKey\_Set \ \wedge \ mId \in memberKey\_Set$

THEN
$reservation := reservation - \{(bId, mId)\}$ ||
$position := position \lhd\!\!+ \{(x_1, x_2, y) \mid x_1 = bId \wedge y = position(bId, x_2) - 1 \wedge x_2 \neq mId \wedge x_2 \in membRes(bId) \wedge position(bId, mId) < position(bId, x_2)\}$
END
END

Let us recall that we cannot prove the consistency of this B machine since the operation preconditions are only typing constraints and therefore are not sufficient to discharge the generated proof obligations.

4.3 Static Part and Signature of Operations

To syntactically distinguish identifiers of the EB^3 specification from those of the B specification, we denote by $id^\star$ the B identifier corresponding to identifier id in EB^3:

- state variable $b^\star$ corresponds to attribute b,
- type $T^\star$ corresponds in B to type T from EB^3,
- operation $a^\star$ is the B translation of action `a`,
- each formal parameter q_i from action `a` is mapped to its corresponding formal parameter $q_i^\star$ in operation $a^\star$.

The static part of B specifications is obtained by translation from the requirements class diagram. The rules are presented in [FLN96, Ngu98]. For the sake of concision, we present in this paper all the B specifications in the same B machine.

Each recursive function k defining a key in the EB^3 specification corresponds to a state variable $k^\star$ in B. The invariant for $k^\star$ is an inclusion of the form $k^\star \subseteq T^\star$, where $T^\star$ represents the set of all the possible values of k. For instance, the key of entity type `book` is `bookKey`. It corresponds in B to state variable $bookKey$. We denote by $bookKey\_Set$ the set of all the possible values for $bookKey$. The invariant property for $bookKey$ is then:

$$bookKey \subseteq bookKey\_Set$$

Each non-key attribute b of the class diagram corresponds to a state variable $b^\star$. The invariant is of the form $b^\star \in k^\star \rightarrow T^\star$, where $k^\star$ is the state variable that corresponds to the key of the entity type or the association in which b is defined and $T^\star$ represents the set of all the possible values of b. The signature of attribute definitions provides the types that are not specified by the user requirements class diagram. For instance, type $Title\_Type$ of attribute $title$ in entity type `book` is provided by the signature of its attribute definition:

$$title(bId : bookKey\_Set) : Title\_Type$$

The invariant of variable $title$ in B is then:

$$title \in bookKey \rightarrow Title\_Type$$

We consider only binary associations, because n-ary associations can always be decomposed into $n-1$ binary associations with additional constraints. Each association ass corresponds to a state variable $ass^\star$ that is a relation between the key state variables of the two entity types that compose the association. According to the multiplicity of ass, $ass^\star$, or its inverse, is defined by a partial function, or an injection, etc. For instance, an association *,* is represented in B by a relation ($\leftrightarrow$), while an association *,0..1 is a partial function ($\nrightarrow$). The whole set of rules is presented in [Ngu98].

An association ass determines two roles. A role r gives for each entity the entity or the set of entities linked to it by ass. For instance, role `borrower` represents the borrower of a book. Each role is translated into B by a definition.

For each action defined in EB^3, an operation is created in B. The signature of B operations comes from the signature of actions in the EB^3 specification. Thus, EB^3 action

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

is translated in B as an operation of the following signature:

$$\begin{array}{l} q^\star_{n+1}, \ldots, q^\star_m \longleftarrow \mathbf{a}^\star(q^\star_1, \ldots, q^\star_n) \triangleq \\ \quad \text{PRE} \;\; q^\star_1 \in T^\star_1 \;\; \wedge \;\; \ldots \;\; \wedge \;\; q^\star_n \in T^\star_n \\ \quad \text{THEN} \;\; \ldots \\ \quad \text{END;} \end{array}$$

When the output parameter type of action `a` is `void`, then the operation has no output in B. Preconditions also come from the signature of actions.

4.4 Substitutions for Initialization

For each attribute definition, there exists an input clause of the form $\perp : u$, possibly by default. It denotes the initial value of the attribute. Indeed, when $last(s) = \perp$, the system trace s is empty; it denotes the case where the system has just been started and no input has been received yet. Thus, the empty trace describes the initial state of the system.

Let $b_1, \ldots, b_m$ be the attribute definitions of the EB^3 specification. For each b_i, a substitution S_i is generated for the initialization. Then, the initialization of the B machine is given by the parallel composition of all the S_i:

$$\begin{array}{l} \text{INITIALISATION} \\ \quad S_1 \;||\; S_2 \;||\; \ldots \;||\; S_m \end{array}$$

The initialization substitution S_i for an attribute definition b_i is generated as follows.

Key Attributes

Expression u in input clause $\perp : u$ of a key attribute definition is the value of the state variable corresponding to the key. Let us recall that a key is formed by one or several key attributes. Let k a key and T_k the type of its key values.

The most common value for u in a key definition is $\emptyset$, which means that there are no entities of the corresponding entity type. Then, the substitution for initialization is $k^\star := \emptyset$.

The other possible value for u is an enumerated set $\{c_1, \ldots, c_n\}$. It corresponds to the creation of n instances of the key. Then, the substitution in B is $k^\star := \{c_1^\star, \ldots, c_n^\star\}$, where all the elements of the set are constants of type $T_k^\star$ defined in the B machine.

Non-Key Attributes

Expression u in input clause $\perp : u$ of a non-key attribute provides attribute values for some key values. By the consistency condition, these key values must exist. This means that they have been created at the initialization. Let b be a non-key attribute associated to key k with values of type T_b. Then, $b^\star$ is a total function from $k^\star$ to $T_b^\star$.

The most common value for u is $\perp$. In that case, any value is defined for b. Then, the initialization in B is $b^\star := \emptyset$.

The other possible value for u is a conditional term of the form

$$
\begin{array}{l}
\textbf{if } k = c_1 \textbf{ then } d_1 \\
\textbf{else if } k = c_2 \textbf{ then } d_2 \\
\qquad \textbf{else } \ldots \\
\qquad \textbf{end} \\
\textbf{end},
\end{array}
$$

where $c_1, \ldots, c_n$ are possible key values for key attribute k of b and $d_1, \ldots, d_n$ are constants of type T_b. Then, the initialization substitution in B is $b^\star := \{c_1^\star \mapsto d_1^\star, \ldots, c_n^\star \mapsto d_n^\star\}$, where $c_1^\star, \ldots, c_n^\star$ are constants already defined by the initialization of key attributes and $d_1^\star, \ldots, d_n^\star$ are other constants of type $T_b^\star$ defined in the B machine.

4.5 Substitution for an Operation

Once all the substitutions S_b have been generated for each attribute b of $B(a)$, then operation a is specified in B. If $B(a) = \{b_1, \ldots, b_m\}$, then $a^\star$ is defined as follows:

$$
\begin{array}{l}
p_{n+1}^\star, \ldots, p_m^\star \longleftarrow \mathbf{a}^\star(p_1^\star, \ldots, p_n^\star) \triangleq \\
\quad \text{PRE } \ p_1^\star \in T_1^\star \ \wedge \ \ldots \ \wedge \ p_n^\star \in T_n^\star \\
\quad \text{THEN } \ S_{b_1} \ || \ \ldots \ || \ S_{b_m} \\
\quad \text{END;}
\end{array}
$$

Chapter 5

Synthesizing Substitutions by Using Substitution Formulas

To generate substitutions for the operations of the B machine, we must analyse the input clauses of EB^3 attribute definitions to determine for each action a of the system:

- which attributes are affected by the execution of a;
- what is the effect of a on these attributes.

In this chapter, we use *substitution formula* to synthesize operation substitutions. This notion is described in Sect. 5.3. The general algorithm is the following:

For each action a of the EB^3 specification
    determine attributes $B(a)$ that are affected by the reception of a
    for each attribute b of $B(a)$
        determine input clauses $IC_b(a)$ of the form $a(\overrightarrow{p}) : u$
        for each input clause $a(\overrightarrow{p}) : u$ of $IC_b(a)$
            identify the free variables of $\overrightarrow{p}$ with the formal parameters of a
            identify the hypotheses under which u is valid
        generate the decision tree for b
        generate the substitution formula for $b^\star$
    generate the substitution for operation $a^\star$

An attribute b may be affected by action a if there exists at least one input clause of the form $a(\overrightarrow{p}) : u$. The set of such attributes is $B(a)$. For each b from $B(a)$, there may be several input clauses with label a. The list of these input clauses is $IC_b(a)$.

Let b be an attribute of $B(a)$. Since the first input clause that evaluates to true is the one to be executed, analysis of input clauses of $IC_b(a)$ is done in their declaration order. For each input clause of $IC_b(a)$ of the form $a(\overrightarrow{p}) : u$, we have to generate:

- a substitution θ_u that links each actual parameter of input clause $a(\overrightarrow{p})$ to a formal parameter of action a of the EB^3 specification,
- a conjunction of hypotheses H_u under which u is valid.

Then, to generate the B substitution for attribute b, we build a decision tree whose leaves are functional terms.

5.1 Free Variables and Hypotheses

Let $a(\overrightarrow{p}) : u$ be an input clause of $IC_b(a)$. In this section, we deal with the definition of θ_u and H_u and we show how to identify the actual parameters of input clause a with the formal parameters of action a. Let us recall that $var(u) \subseteq var(\overrightarrow{p}) \cup \overrightarrow{k}$. The identification will be used to determine some variables of $\overrightarrow{k}$ and u from the formal parameters of action a. The principle is similar to the pattern matching described in Sect. 3.4.

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. H_u is initialized to $true$ and θ_u to the empty set. There are three cases to consider:

1. when p_j is the wildcard symbol '_', then u is valid for every value of parameter q_j. θ_u and H_u are not changed;
2. when p_j is a ground term c, then u is valid only for $q_j = c$. Condition $q_j = c$ is then added to hypothesis H_u. θ_u is not changed;
3. when p_j is a variable v, we can associate parameter p_j to parameter q_j and $p_j := q_j$ is added to θ_u; however, if this variable v occurs in more than one place in $p_1, \ldots, p_m$, then we must check that the corresponding parameters in $q_1, \ldots, q_m$ are all the same. In other words, let J_v be the list of indices where v occurs in $p_1, \ldots, p_m$, then, for each $j \in J_v$, substitution $p_j := q_j$ is added to θ_u and condition $\bigwedge_{j_1, j_2 \in J_v} (q_{j_1} = q_{j_2})$ is added to hypothesis H_u.

Let q_1, q_2 and q_3 be the formal parameters of action Transfer. The identification has the following results for input clause Transfer$(bId, mId', _)$ in attribute $nbLoans$:

$$\begin{aligned} \theta_u &= \{bId := q_1, mId' := q_2\} \\ H_u &= true \end{aligned}$$

For input clauses Lend$(bId, mId, Permanent)$ and Lend$(bId, mId, Classic)$ in attribute definition $dueDate$, the results are, respectively,

$$\begin{aligned} \theta_{u_1} &= \{bId := q_1, mId := q_2\} \\ H_{u_1} &= (q_3 = Permanent) \end{aligned}$$

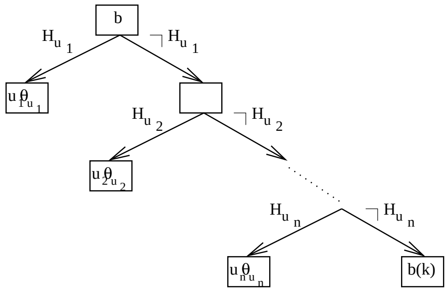

Figure 5.1: Step 1 of the decision tree construction

and

$$\begin{aligned} \theta_{u_2} &= \{bId := q_1, mId := q_2\} \\ H_{u_2} &= (q_3 = Classic) \end{aligned}$$

θ_u is used in the remainder of this report to link the free variables in $var(\overrightarrow{p})$ to the formal parameters of operation $a^\star$, since we consider that the signatures of action a and operation $a^\star$ are the same.

5.2 Construction of the Decision Tree

Let b be a key or a non-key attribute of $IC_b(a)$. Once θ_{u_j} and H_{u_j} have been determined for each input clause $a(\overrightarrow{p_j}) : u_j$ of $IC_b(a)$, then a *decision tree* is built for b. The decision tree is a binary tree generated as follows.

Step 1

The number n of levels of the tree is the number of input clauses of $IC_b(a)$. The leaves of the tree are the input clauses $a(\overrightarrow{p_j}) : u_j$ of $IC_b(a)$, except for the rightmost leaf which is $b(\overrightarrow{k})$. Step 1 is illustrated by Fig. 5.1. The root is b, its left child is $u_1\ \theta_{u_1}$ whose edge is labelled by H_{u_1} and its right child is either the subtree generated at the next level or $b(\overrightarrow{k})$ if there is only one input clause. The edge on the right is labelled by $\neg H_{u_1}$. The subtrees are generated as follows. At level $j-1 \neq n$, the left child is $u_j\ \theta_{u_j}$ labelled by H_{u_j} and the right child is the subtree of level j labelled by $\neg H_{u_j}$. At level n, the subtree is simply leaf $b(\overrightarrow{k})$.

Step 2

Each leaf is of the form $u_j\ \theta_{u_j}$. A leaf with a conditional term is then replaced by a decision subtree whose leaves are all functional terms. The decision subtree for conditional term u_j is generated as follows. The root is $u_j\ \theta_{u_j}$. The children of a node are its **then** and **else** subterms, whose respective edges are labelled by c and $\neg c$. At the end, all the leaves of the tree must be functional terms. For example, the decision tree of input clause Transfer in attribute *nbLoans* is represented in Fig. 5.2.

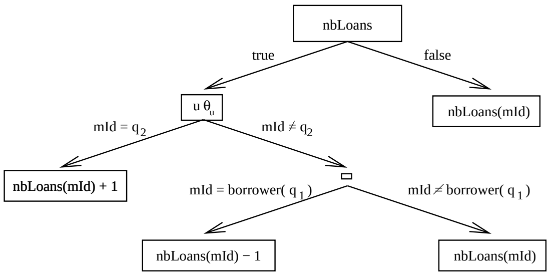

Figure 5.2: The decision tree of input clause Transfer in *nbLoans*

5.3 Substitutions Formulas

The decision tree of b is analysed to generate a *substitution formula* that defines the whole set of substitutions for $b^\star$.

5.3.1 Key Attributes

Let b be a key of $B(a)$. The interesting paths in the decision tree of b are those that start from the root and that lead to a leaf which is distinct of a recursive call to b. Let $u_{j,i}$ be the functional term of the i-th leaf $u_{j,i}\ \theta_{u_j}$ of the subtree corresponding to the j-th input clause of $IC_b(a)$ of attribute b. Let us note $\Phi_{u_{j,i}}$ the conjunction of conditions ϕ labelling the path's edges leading to functional term $u_{j,i}$. Then, the substitution formula S for $b^\star$ is:

$$S = \bigvee_{i,j} ((\Phi_{u_{j,i}} \wedge b^\star := u_{j,i})\theta_{u_j}) \tag{5.1}$$

For instance, the substitution formula for input clause Transfer$(bId, mId, _)$ in key definition *loan* of association `loan` is:

$$b^\star := (loan - \{(bId, borrower(bId))\} \cup \{(bId, mId)\})\theta_u \tag{5.2}$$

Substitution θ_u is $\{bId := bId^\star, mId := mId^\star\}$.

5.3.2 Non-Key Attributes

Let b a non-key attribute associated to key k. The interesting paths are those that start from the root and that lead to a leaf which is distinct of a recursive call $b(\overrightarrow{k})$. Let $u_{j,i}$ be the functional term of the i-th leaf $u_{j,i}\ \theta_{u_j}$ of the subtree corresponding to the j-th input clause of $IC_b(a)$ of attribute b. Let us note $\Phi_{u_{j,i}}$ the conjunction of conditions ϕ labelling the path's edges leading to functional term $u_{j,i}$. Then, the substitution formula for $b^\star$ is the following.

$$b^\star := (A \mathbin{\lhd\!\!\!-} b^\star) \mathbin{\lhd\!\!\!+} R \tag{5.3}$$

where A is the following set defined for every $u_{j,i} = \bot$:

$$A = \{c \mid \exists\, \overrightarrow{k} \cdot \bigvee_j (c = (\overrightarrow{k})\theta_{u_j} \ \ \wedge\ \ \bigvee_i \Phi_{u_{j,i}}\theta_{u_j})\}$$

and R is defined for every $u_{j,i} \neq \perp$ by:

$$R = \{(c, d) \mid \exists \overrightarrow{k} \cdot \bigvee_j (c = (\overrightarrow{k})\theta_{u_j} \ \wedge \ \bigvee_i (\Phi_{u_{j,i}} \wedge d = u_{j,i})\theta_{u_j})\}$$

A represents the attribute values that become undefined for $\overrightarrow{k}$, while R is the set of maplets that assigns the new values of b. Let us note that c can be a vector $\overrightarrow{c} = (c_1, \ldots, c_m)$, where m is the number of key attributes that compose key k.

5.3.3 Generating B Substitutions

We now show how to rewrite the substitution formulas (5.1) and (5.3) as usual B substitutions. Substitution formula S (5.1) and sets A and R from substitution formula (5.3) represent the different B substitutions for attribute b. We use the following rules to rewrite substitution formulas as **IF THEN ELSE** substitutions that involve expressions depending only on state variables and constants defined in the B machine.

Rule 1 (Application of the θ_{u_j}s) *Let Set be of the form S, R or A. For each j, substitution θ_{u_j} is applied to the corresponding expression $expr_j = expr\ \theta_{u_j}$ in Set. Thus, for each substitution $v := q$ in θ_{u_j}, where v is a variable of Set, then each occurrence of v in $expr_j$ is replaced by q and θ_{u_j} is removed from expression $expr_j$. When Set is of the form A or R, then if v is one of the variables in $\overrightarrow{k}$, expression $\exists v$ is removed for expression $expr_j$.*

S is now of the form $\bigvee_{i,j} SExpr_{i,j}$, where each expression $SExpr_{i,j}$ determines a possible substitution for b. So S is a disjunction of possible substitutions. Conditions $\Phi_{u_{j,i}}$ determine under which conditions such substitutions are valid. The following rule is used to decompose S into a **IF THEN ELSE** expression:

Rule 2 (Decomposition of S) *For each i, j, each condition of the form $q = w$ or $q \in T$ in $SExpr_{i,j}$ generates an expression of the form **IF** cond **THEN** S_1 **ELSE** S_2, where cond is the condition $q = w$ or $q \in T$, S_1 is the set S restricted to the expressions $SExpr_{j,i}$ for which condition cond is true, and S_2 is the set S restricted to the expressions $SExpr_{j,i}$ for which condition cond is false. Any occurrence of cond and $\neg$cond is then removed from S_1 and S_2.*

The resulting expression is a **IF THEN ELSE** expression for key definitions.

Let us now consider *Set* be of the form A or R. Existential quantification in sets of the form A or R are simplified by the following rule. We suppose that c is of the form $c = (c_1, \ldots, c_m)$.

Rule 3 (One-point rule) *Let Set be of the form A or R. For each condition $c_l = w$, where w is a bound variable of Set or a formal parameter of action* a*, then,*

- *each occurrence of w in Set is replaced by c_l,*
- *condition $c_l = w$ is removed from Set,*
- *if w is one of the variables in $\overrightarrow{k}$, then expression $\exists w$ is removed from Set.*

Sets A and R are now of the form $\{e \mid \bigvee_{i,j} expr_{i,j}\}$, where each expression $expr_{i,j}$ determines one or more elements e in the set. There are two kinds of conditions on c in each expression $expr_{j,i}$: equalities of the form $c = w$ and inclusions of the form $c \in T$. Since conditions $\Phi_{u_{j,i}}$ have been generated from a decision tree, there are also negations of such conditions.

Conditions $\Phi_{u_{j,i}}$ not only determine which elements e are defined in Set, but also under which conditions such elements are valid. This means that some conditions in $\Phi_{u_{j,i}}$ are constraints on the formal parameters of action a, and consequently on formal parameters of operation $a^\star$. Such conditions are of the form $q = w$ or $q \in T$, where q is a formal parameter of a. For instance, hypothesis H_u defined in Sect. 5.1 is a condition of this kind. The following rule allows us to decompose Set into several sets Set_l, each one associated to a condition $cond_l$.

Rule 4 (Detection of Hypotheses) *Let Set be of the form A or R. For each j, for each i, each condition of the form $q = w$ or $q \in T$ in $expr_{i,j}$ generates an expression of the form **IF** cond **THEN** Set_1 **ELSE** Set_2, where cond is the condition $q = w$ or $q \in T$, Set_1 is the set Set restricted to the expressions $expr_{j,i}$ for which condition cond is true, and Set_2 is the set Set restricted to the expressions $expr_{j,i}$ for which condition cond is false. Any occurrence of cond and $\neg$cond is then removed from Set_1 and Set_2.*

The application of rule (4) is simplified by the general form of conditions $\Phi_{u_{j,i}}$. If expression $expr_{j,i}$ contains a condition on q, then there exists an expression $expr_{j',i'}$ with the negation of this condition. One can also build a decision tree with only the conditions on q as labels to retrieve the nested above-mentionned **IF THEN ELSE** expressions.

Sets Set_l in the resulting **IF THEN ELSE** expression are then simplified to be transformed into enumerated sets. For each Set_l, the remaining conditions in $\Phi_{u_{j,i}}$ are conditions on c for sets of the form A or R. Equalities on c can be propagated to the other equations in $expr_{i,j}$ with the following rule.

Rule 5 (Propagation of c) *Let Set_l be of the form R. For each $expr_{i,j}$ in Set_l, for each equation of the form $d = u_{j,i}$ in $expr_{i,j}$, where $u_{j,i}$ is an expression depending on c and if c is determined in expression $expr_{i,j}$ by a condition of the form $c = f(t)$, where f is an operator or an attribute, and t is a formal parameter of action a, then each occurrence of c in $d = u_{j,i}$ is replaced by $f(t)$.*

At this point c and d are determined by the remaining conditions. Each Set_l is decomposed into the union of the sets corresponding to each branch of $\bigvee_{i,j}$.

Rule 6 (Disjunction of sets) *Let Set_l be of the form $\{e \mid \bigvee_k expr_k\}$, then Set_l is transformed into $\cup_k\{e \mid expr_k\}$.*

Some sets S_{l_k} of the form $\{e \mid expr_k\}$ can be transformed into enumerated sets if d depends only on formal parameters of a.

Rule 7 (Transformation into enumerated set) *Let Set_{l_k} be of the form $\{e \mid expr_k\}$, where $expr_k$ contains an equation of the form $d = expr$, where expr only depends on the formal parameters of a, then e is determined. Such sets Set_{l_k} that correspond to a singleton are grouped into an enumerated set that collects the elements from each singleton.*

The resulting expression is a **IF THEN ELSE** substitution in B.

5.3.4 Examples

An example of substitution formula for key definitions has already been presented in Sect. 5.3.1:

$$b^\star := (loan - \{(bId, borrower(bId))\} \cup \{(bId, mId)\})\theta_u \qquad (5.4)$$

It is derived from (5.1) for input clause $\mathsf{Transfer}(bId, mId, \_)$ in key definition $loan$ of association `loan`. The corresponding substitution θ_u is $\{bId := bId^\star, mId := mId^\star\}$. Then, by applying rule (1),

$$b^\star := loan - \{(bId^\star, borrower(bId^\star))\} \cup \{(bId^\star, mId^\star)\}$$

Let us now consider non-key attribute definitions. When u is $\perp$, then attribute b becomes undefined at $\overrightarrow{k}$. For instance, action $\mathsf{Discard}(bId)$ sets the value of attribute $title$ to $\perp$ for book entity bId. The corresponding substitution formula is derived from (5.3):

$$b^\star := A \lhd b^\star$$

where A is defined as follows.

$$A = \{c \mid \exists\, bId \cdot (c = bId)\ \theta_u \ \ \wedge \ \ true\ \theta_u\}$$

Substitution θ_u is simply $\{bId := bId^\star\}$. Consequently, by application of rule (1),

$$A = \{bId^\star\}$$

The corresponding substitution in operation **Discard** is then:

$$title := \{bId\} \lhd title$$

When u is a conditional term, then the substitution formula (5.3) takes all the constraints on the input parameters of the B operation and on the key values into account. For instance, the conditional term for input clause $\mathsf{Transfer}$ in attribute definition $nbLoans$ involves two instances of attribute key mId: mId' and $borrower(bId)$. The generated substitution formula is the following.

$$b^\star := b^\star \lhd\!\!+ R$$

where R is defined by:

$$\begin{aligned} R \quad = \quad & \{(c, d) \mid \exists\, mId \cdot (c = mId)\ \theta_u \ \ \wedge \\ & ((mId = mId' \ \ \wedge \ \ d = nbLoans(mId) + 1)\ \theta_u) \vee \\ & ((mId \neq mId' \ \ \wedge \ \ mId = borrower(bId) \wedge d = nbLoans(mId) - 1)\ \theta_u)\} \end{aligned}$$

Substitution θ_u is here $\{bId := bId^\star, mId' := mId^\star\}$. By application of rule (1),

$$\begin{aligned} R \quad = \quad & \{(c, d) \mid \exists\, mId \cdot c = mId \ \ \wedge \\ & (mId = mId^\star \ \ \wedge \ \ d = nbLoans(mId) + 1) \vee \\ & (mId \neq mId^\star \ \ \wedge \ \ mId = borrower(bId^\star) \wedge d = nbLoans(mId) - 1)\} \end{aligned}$$

Then, we use the one-point rule (3) to simplify the existential quantification:

$$\begin{aligned} R \;=\; & \{(c,d) \mid \\ & (c = mId^\star \;\wedge\; d = nbLoans(c) + 1) \vee \\ & (c \neq mId^\star \;\wedge\; c = borrower(bId^\star) \wedge d = nbLoans(c) - 1)\} \end{aligned}$$

Since there is no hypothesis on $mId^\star$ or $bId^\star$ in R, we can directly use rule (5) to propagate the equalities on c:

$$\begin{aligned} R \;=\; & \{(c,d) \mid \\ & (c = mId^\star \;\wedge\; d = nbLoans(mId^\star) + 1) \vee \\ & (c \neq mId^\star \;\wedge\; c = borrower(bId^\star) \wedge d = nbLoans(borrower(bId^\star)) - 1)\} \end{aligned}$$

Rules (6) and (7) are then applied to obtain:

$$R \;=\; \{(mId^\star, nbLoans(mId^\star) + 1), (borrower(bId^\star), nbLoans(borrower(bId^\star)) - 1)\}$$

The resulting B substitution for operation $Transfer$ is:

$$\begin{aligned} nbLoans := nbLoans \lhd\!\!\!+ \{& mId \mapsto nbLoans(mId) + 1, \\ & borrower(bId) \mapsto nbLoans(borrower(bId) - 1\} \end{aligned}$$

Let us recall that c may be a vector. For instance, if we consider attribute $dueDate$ in association `loan`, then c is of the form $c = (bId, mId)$ in the corresponding substitution formula. Substitution formulas also take conditions on the formal parameters of the action into account. Such conditions may appear in input clause $a(\overrightarrow{p}) : u$ either in $a(\overrightarrow{p})$ when one of the input parameters of a is instantiated by a value or in expression u when a conditional term involves a condition on the formal parameters of a. For instance, input clauses $\mathsf{Lend}(bId, mId, Permanent)$ and $\mathsf{Lend}(bId, mId, Classic)$ from attribute definition $dueDate$ in association `loan` are examples for the first case. Input clause $\mathsf{Transfer}$ in attribute definition $dueDate$ is an example for the second case.

The substitution formula for Lend in $dueDate$ is the following:

$$b^\star := b^\star \lhd\!\!\!+ R$$

where R is defined by:

$$\begin{aligned} R \;=\; & \{(c,d) \mid \exists (bId, mId) \cdot (c = (bId, mId)\ \theta_{u_1} \;\wedge \\ & ((typeOfLoan^\star = Permanent \;\wedge\; d = CurrentDate + 365)\ \theta_{u_1})) \vee \\ & (c = (bId, mId)\ \theta_{u_2} \;\wedge\; ((typeOfLoan^\star \neq Permanent \;\wedge \\ & typeOfLoan^\star = Classic \wedge d = CurrentDate + loanDuration(mId))\ \theta_{u_2}))\} \end{aligned}$$

where $\theta_{u_1} = \theta_{u_2} = \{bId := bId^\star, mId := mId^\star\}$. By application of rule (1),

$$\begin{aligned} R \;=\; & \{(c,d) \mid (c = (bId^\star, mId^\star) \;\wedge \\ & (typeOfLoan^\star = Permanent \;\wedge\; d = CurrentDate + 365)) \vee \\ & (c = (bId^\star, mId^\star) \;\wedge\; (typeOfLoan^\star \neq Permanent \;\wedge \\ & typeOfLoan^\star = Classic \wedge d = CurrentDate + loanDuration(mId^\star)))\} \end{aligned}$$

Let us note that the existential quantification is already removed. Rule (4) can now be applied since there are conditions on formal parameter $typeOfLoan^\star$

of action Lend. The first condition $typeOfLoan^\star = Permanent$ generates the following **IF THEN ELSE** expression:

$$
\begin{aligned}
R \quad = \quad & \textbf{IF } typeOfLoan^\star = Permanent \textbf{ THEN} \\
& \{(c,d) \mid c = (bId^\star, mId^\star) \ \wedge \ d = CurrentDate + 365\} \\
& \textbf{ELSE} \\
& \{(c,d) \mid c = (bId^\star, mId^\star) \ \wedge \\
& typeOfLoan^\star = Classic \wedge d = CurrentDate + loanDuration(mId^\star)\}
\end{aligned}
$$

Rule (4) is applied once again for condition $typeOfLoan^\star = Classic$ in the **ELSE** part. As a matter of fact, it is not necessary since parameter $typeOfLoan^\star$ can only have two values, *Permanent* and *Classic*, but this cannot be simply detected automatically. By rules (6) and (7) applied on each of the **THEN** and **ELSE** parts, we deduce:

$$
\begin{aligned}
R \quad = \quad & \textbf{IF } typeOfLoan^\star = Permanent \textbf{ THEN} \\
& \{((bId^\star, mId^\star), CurrentDate + 365)\} \\
& \textbf{ELSE} \\
& \{((bId^\star, mId^\star), CurrentDate + loanDuration(mId^\star))\}
\end{aligned}
$$

The B substitution for attribute *dueDate* in operation *Lend* is then:

IF $typeOfLoan = Permanent$
THEN $dueDate := dueDate \lhd\!\!- \{(bId, mId) \mapsto CurrentDate + 365\}$
ELSE $dueDate := dueDate \lhd\!\!- \{(bId, mId) \mapsto CurrentDate + loanDuration(mId)\}$
END

5.4 Conclusion

In this chapter, we have first shown how to generate substitution formulas for attribute definitions and then how to synthesize usual B substititions from substitution formulas. This approach allows us to describe different B substitutions in a compact form and makes the translation from EB^3 into B easier. Nevertheless, this requires the existence of a tool that is able to interpret the rules presented in Sect. 5.3.3 in order to rewrite substitution formulas as B substitutions. As a future work, we aim at implementing an interpreter for such formulas.

Chapter 6

Synthesizing Substitutions Directly

In the previous chapter, we have shown how to generate substitution formulas that characterize all the B substitutions for an attribute. In this chapter, we aim at directly generating explicit B substitutions without using substitution formulas. The algorithm is the following:

For each action a of the EB^3 specification
    determine attributes $B(a)$ that are affected by the reception of a
    for each attribute $b(\overrightarrow{k})$ of $B(a)$
        determine input clauses $IC_b(a)$ of the form $a(\overrightarrow{p}) : u$
        for each input clause $a(\overrightarrow{p}) : u$ of $IC_b(a)$
            identify the free variables of $\overrightarrow{p}$ with the formal parameters of a
            if $\overrightarrow{k}$ can be determined from $\overrightarrow{p}$
            then generate the substitution for $b(s \leftarrow a, \overrightarrow{k})$
            else
                determine $\overrightarrow{k}$ from $\overrightarrow{p}$ and the conditions on $\overrightarrow{k}$
                for each different case of condition on $\overrightarrow{k}$
                    generate the substitution for $b(s \leftarrow a, \overrightarrow{k})$
            end
        generate the substitution for $b^\star$
    generate the substitution for operation $a^\star$

For each input clause of $IC_b(a)$ of the form $a(\overrightarrow{p}) : u$, where u is a functional term, we have to generate:

- a mapping θ_u that associates each actual parameter of input clause $a(\overrightarrow{p})$ to a formal parameter of action a of the EB^3 specification,
- a substitution S_u in B that corresponds to the computation of u,
- a conjunction of hypotheses H_u under which u is valid.

θ_u and H_u are generated as in Sect. 5.1. θ_u is here considered as a mapping, so each assignment $:=$ in Sect. 5.1 is replaced by an equality $=$.

When u is a conditional term, then we have also to generate one or more functional terms u_j, where each u_j is associated to a mapping θ_{u_j}, a B substitution S_{u_j} and a conjunction of hypotheses H_{u_j}. Section 6.2.4.1 shows how to use all these elements to generate the B substitution for attribute b.

Once θ_u has been generated for input clause $a(\overrightarrow{p}) : u$ as in Sect. 5.1, then a key value of b can be partly computed, since we can determine the variables in $\overrightarrow{k} \cap var(\overrightarrow{p})$ from the formal parameters of action a. It remains to find how to evaluate variables in $\overrightarrow{k} - var(\overrightarrow{p})$, which we denote in the sequel by $\overrightarrow{k_\sharp}$. If $\overrightarrow{k_\sharp}$ is empty, then a key value of $\overrightarrow{k}$ can be determined. If b is a key or if u is a functional term, then $\overrightarrow{k_\sharp}$ is always empty. If $\overrightarrow{k_\sharp}$ is not empty, then u must be a conditional term to determine the missing variables of $\overrightarrow{k_\sharp}$.

6.1 Functional Terms

Let $a(\overrightarrow{p}) : u$ be an input clause of $IC_b(a)$, where u is a functional term. In that case, all the free variables of $\overrightarrow{p}$ are identified with formal parameters of action a, thanks to substitution θ_u. Substitution S_u is generated as follows.

6.1.1 Key Attributes

Let k be a key of $B(a)$ with values of type T_k. The possible expressions for u in input clause $a(\overrightarrow{p}) : u$ are:

- $k(front(s)) \cup S$, which denotes the addition of new key values,
- $k(front(s)) - S$, which denotes the removal of key values,
- $k(front(s)) \cup S - S'$, if k is the key of an association.

Then, u denotes the new value of k and is directly specified in B as a substitution for state variable $k^\star$ with the corresponding sets $S^\star$:

$$k^\star := k^\star \cup S^\star$$
$$k^\star := k^\star - S^\star$$
$$k^\star := k^\star \cup S^\star - S'^\star$$

$S^\star$ (or $S'^\star$) is generated as follows. S is a set composed of constants of type T_k and/or variables from $\overrightarrow{p}$. Each constant c from S corresponds to a constant $c^\star$ of type $T_k^\star$ that is defined in B. Each variable v is linked to a formal parameter of $a^\star$ by substitution θ_u. $S^\star$ is then the set of such elements $c^\star$ and $v^\star$ that correspond to elements c and v from set S. Then, $S^\star = S\ \theta_u$.

For instance, substitution $memberKey := memberKey \cup \{mId\}$ in operation **Register** comes from input clause $\mathsf{Register}(mId, \_) : memberKey() \cup \{mId\}$ from attribute definition of key $memberKey()$.

6.1.2 Non-Key Attributes

Let b be a non-key attribute of $B(a)$ associated to key k with values of type T_b. Thus, state variable $b^\star$ is a total function from $k^\star$ to $T_b^\star$. In this paragraph, we deal with non-key attributes when $\overrightarrow{k_\sharp}$ is empty. This means that a key value $\overrightarrow{k_v}$

of attribute b can be computed for $\overrightarrow{k}$ with the variables of $\overrightarrow{p}$ and substitution θ_u.

For each kind of input clause, the key value $\overrightarrow{k_v}$ must be translated in B to modify state variable $b^\star$. Each k_j in $\overrightarrow{k}$ is a variable that depends on the free variables of $\overrightarrow{p}$. And it is the substitution θ_u that determines the key value $\overrightarrow{k_v}$. So, each variable k_j is bound to a formal parameter of action a by θ_u and, consequently, to a formal parameter $v^\star$ of operation $a^\star$. Since all the variables of $\overrightarrow{k}$ can be translated in B, we note $\overrightarrow{k}^\star$ the result of this translation.

The possible values of u in input clause $a(\overrightarrow{p}) : u$ are now:

- $\perp$, an undefined value for b,
- c, a constant from type T_b,
- v, a variable from $var(\overrightarrow{p}) \cup var(\overrightarrow{k})$,
- $g(\overrightarrow{t})$, where each $t_i \in \overrightarrow{t}$ is a functional term and g is either an attribute or an operator.

When u is $\perp$, then the key value of $\overrightarrow{k}$ has no value anymore for attribute b. The corresponding substitution in B is:

$$b^\star := \{\overrightarrow{k}^\star\} \lhd\!\!\!- \, b^\star$$

For instance, action Unregister(mId) sets the value of attribute $nbLoans$ to $\perp$ for member entity mId. The corresponding substitution in operation **Unregister** is:

$$nbLoans := \{mId\} \lhd\!\!\!- \, nbLoans$$

When u is a constant c or a variable v, then key value $\overrightarrow{k}$ is associated to its new value. Variable v is determined, because $\overrightarrow{p}$ and $\overrightarrow{k}$ are known. We use θ_u to retrieve input parameter $v^\star$ of $a^\star$. The substitutions are then:

$$b^\star := b^\star \Leftarrow\!\!\!+ \{\overrightarrow{k}^\star \mapsto c^\star\}$$
$$b^\star := b^\star \Leftarrow\!\!\!+ \{\overrightarrow{k}^\star \mapsto v^\star\}$$

For a new title of a book given by action Modify in attribute $title$, the corresponding substitution is:

$$title := title \Leftarrow\!\!\!+ \{bId \mapsto newTitle\}$$

When u is of the form $g(\overrightarrow{t})$, then it is always of type T_b. This expression corresponds to a computation that outputs a value from type T_b. It may involve recursive calls of b and/or calls to other attribute definitions. However, input parameters of these calls are known from $\overrightarrow{p}$ and $\overrightarrow{k}$. We suppose that operators g have an implementation $g^\star$ in B with the corresponding data types of the attribute definitions. Then, the substitution is:

$$b^\star := b^\star \Leftarrow\!\!\!+ \{\overrightarrow{k}^\star \mapsto g^\star(\overrightarrow{t}^\star)\}$$

where $\overrightarrow{t}^\star$ denotes the application of θ_u for each t_i in $\overrightarrow{t}$.

For instance, consider the input clause of action Take in attribute definition of $nbLoans$

$$\mathsf{Take}(\_, mId, \_)\ :\ 1\ +\ nbLoans(mId),$$

The key value is here mId which depends on the input mId of action Take. u is of the form:

$$g(mId) = 1 + nbLoans(mId)$$

where $nbLoans$ refers to a recursive call of the attribute definition for trace $front(s)$. Then,

$$g^{\star}(mId) = 1 + nbLoans(mId)$$

where mId is the input parameter of B operation **Take**, $+$ is the addition operator of natural numbers defined in B, and $nbLoans$ is the state variable of type $memberKey \rightarrow NAT$. The corresponding substitution is:

$$nbLoans := nbLoans \mathbin{\lhd\!\!\!-} \{mId \mapsto 1 + nbLoans(mId)\}$$

6.2 Conditional Terms

Let b be a key or a non-key attribute of $B(a)$. Let θ_u be the substitution for input clause $a(\overrightarrow{p})$: u of $IC_b(a)$ (see Sect. 5.1). To determine the remaining free variables $\overrightarrow{k_\sharp}$, we first apply θ_u to u, noted $u\ \theta_u$, in order to propagate the substitutions determined from the pattern condition. A decision tree DT_b is built for attribute b as described in Sect. 5.2.

6.2.1 Decision Paths

Let a *decision path* be a path starting from the root of the decision tree DT_b and leading to a leaf which is distinct of a recursive call $b(\overrightarrow{k})$. There are two kinds of conditions labelling each path's edge. Labels of the form $k_j = w$, where w is a constant or a variable, or $k_j \in S$, where S is a set of constants or variables, are conditions on the free variables $\overrightarrow{k_\sharp}$. Labels of the form $q_j = w$ or $q_j \in S$ are hypotheses on the formal parameters of action a. For each decision path u_i, we have to generate:

- a mapping θ_{u_i} that determines all the free variables of $\overrightarrow{k}$ from the formal parameters of action a,
- a substitution S_{u_i} that computes u,
- a conjunction of hypotheses H_{u_i} under which S_{u_i} is valid.

A decision path of the decision tree represents a formula which is a conjunction of the conditions labelling the path's edge. The formula has the following form.

$$\forall \overrightarrow{k_\sharp} \bullet (\bigwedge_x \phi_x) \Rightarrow (b(\overrightarrow{k}) = u_i)\theta_u \tag{6.1}$$

For instance, the decision tree of Fig. 5.2 represents two formula.

$$\forall mId \bullet mId = q_2 \Rightarrow nbLoans(mId) = nbLoans(mId) + 1 \tag{6.2}$$

$$\begin{array}{l} \forall mId \bullet mId \neq q_2 \wedge mId = borrower(q_1) \\ \Rightarrow \\ nbLoans(mId) = nbLoans(mId) - 1 \end{array} \tag{6.3}$$

6.2.2 Identification of the Missing Free Variables

In order to determine the missing free variables of $\overrightarrow{k}$, we suppose that u satisfies the two following conditions. Firstly, for each variable $k \in \overrightarrow{k_\sharp}$, there must be an assignment predicate of the form $k = w$ or $k \in S$ in each decision path, because of the consistency condition. Secondly, there must not be circular references between the variables of $\overrightarrow{k_\sharp}$ in a decision path, when attribute definitions are correctly specified. Consequently, for each decision path, one or more key values of $\overrightarrow{k}$ can be computed in function of the formal parameters of action a.

Let u_i be a decision path of the decision tree. We want to determine the mapping θ_{u_i} of conditions on the free variables $\overrightarrow{k}$. Some free variables $k_1, ..., k_m$ have been identified with θ_u. Each condition of the form $k = w$ or $k \in S$ labelling the edges of decision path u_i determines one of the missing free variables from the formal parameters of action a. Then, θ_{u_i} is the set composed of:

- the elements of θ_u that are of the form $k = w$ or $\neg\ (k = w)$, where k is a free variable of $\overrightarrow{k}$;
- each condition of the form $k = w$, $\neg\ (k = w)$, $k \in S$ or $\neg\ (k \in S)$ belonging to the decision path that leads to u_i, and where k is a free variable of $\overrightarrow{k}$;
- each condition that depends on k, but that is not a condition of the form $q = w$, $\neg\ (q = w)$, $q \in S$ or $\neg\ (q \in S)$, where q is a formal parameter of action a.

As an example, consider the decision paths of the decision tree generated for input clause Transfer in attribute *nbLoans* (see Fig. 5.2). In the first decision path on the left,

$$\theta_{u_1} = \{mId = q_2\}$$

since $\overrightarrow{k} = mId$. In the second decision path,

$$\theta_{u_2} = \{mId \neq q_2, mId = borrower(q_1)\}$$

6.2.3 Generating Substitutions

Once θ_{u_i} is determined, substitution S_{u_i} for decision path u_i is generated as follows. The leaves of the decision tree are all functional terms. Let L_{u_i} denote the leaf of decision path u_i.

When all the conditions of θ_{u_i} are of the form $k = w$, then a key value of $\overrightarrow{k}$ can be computed with the values of formal parameters of a. We suppose that each condition $k = w$ of θ_{u_i} is a substitution $k := w$. Each w_j in $k_j = w_j$ is of one of the following forms:

- q, a formal parameter of action a,
- $f(\overrightarrow{t})$, where f is an attribute or an operator and each t_i in $\overrightarrow{t}$ is either a constant c or a variable v.

Then, each q is associated to an input parameter $q^\star$ of $a^\star$ in B. Each c of type T is associated to a constant $c^\star$ of type $T^\star$ in B. Each variable v is bound to a formal parameter of action a by θ_u or θ_{u_i} and then, consequently, to a formal parameter $v^\star$ of operation $a^\star$. An attribute f is associated to state variable $f^\star$. We suppose that an operator $f^\star$ implements in B operator f. Since all the variables of $\overrightarrow{k}$ can be translated in B for decision path u_i, we note $\overrightarrow{k_{u_i}}^\star$ the result of this translation.

Then, the substitution on state variable $b^\star$ corresponding to decision path u_i is:

$$b^\star := \{\overrightarrow{k_{u_i}}^\star\} \lhd b^\star \text{ if } L_{u_i} = \perp$$

or

$$b^\star := b^\star \mathbin{\lhd\!\!+} \{\overrightarrow{k_{u_i}}^\star \mapsto L^\star_{u_i}\}, \text{ otherwise}$$

where $L^\star_{u_i}$ is the translation in B of the functional term L_{u_i}. Since L_{u_i} is already the result of the application of substitution θ_u, we apply substitution θ_{u_i} to translate the free variables of L_{u_i} into formal parameters in B. Constants, attributes and operators are treated as in Sect. 6.1. We note that: $L^\star_{u_i} = L_{u_i}\ \theta_{u_i}$

If one or more conditions in θ_{u_i} are of the form $k \in w$, then there are several possible key values for the same decision path. Conditions of the form $k = w$ in θ_{u_i} are considered as substitutions $k := w$. The set of such substitutions is denoted by θ'_{u_i}. Then, conditions of the form $k \in S$ denote a set of possible elements for variable k. $L^\star_{u_i}$ and $\overrightarrow{k_{u_i}}$ are determined from θ'_{u_i} and from the conditions on k. The equivalent substitution in B is:

$$b^\star := b^\star \mathbin{\lhd\!\!+} X$$

where X is the set of elements $(\overrightarrow{x}, y)$ that satisfy the conditions in $\theta_{u_i} - \theta'_{u_i}$. X is defined as follows.

$$X = \{(\overrightarrow{x}, y) \mid \overrightarrow{x} = \overrightarrow{k_{u_i}}\ \theta'_{u_i} \ \wedge\ y = L_{u_i}\ \theta'_{u_i} \ \wedge\ (\theta_{u_i} - \theta'_{u_i})\}$$

For instance, input clause Cancel in attribute definition *position* of association `reservation` has such a condition on $\overrightarrow{k}$: $mId \in membRes(bId)$. Let us consider the decision path leading to functional term $position(bId, mId) - 1$. Then, θ_{u_i} is:

$$\begin{aligned}\theta_{u_i} \quad = \quad & \{bId = q_1, mId' = q_2, mId \neq q_2, mId \in membRes(bId), \\ & position(bId, mId') < position(bId, mId)\}\end{aligned}$$

where q_1 and q_2 are the formal parameters of operation *Cancel*. Substitution θ'_{u_i} is therefore:

$$\{bId := q_1, mId' := q_2\}$$

Then, X is defined by:

$$\begin{aligned}X \quad = \quad & \{(x_1, x_2, y) \mid x_1 = q_1 \ \wedge\ x_2 = mId \ \wedge\ y = position(q_1, mId) - 1 \ \wedge \\ & mId \neq q_2 \ \wedge\ mId \in membRes(bId) \ \wedge\ position(q_1, q_2) < position(q_1, mId)\}\end{aligned}$$

Let us note mId and bId the formal parameters of operation *Cancel*. Then, the corresponding substitution for *position* is:

$$\begin{aligned}& position := position \mathbin{\lhd\!\!+} \{(x_1, x_2, y) \mid x_1 = bId \wedge y = position(bId, x_2) - 1 \wedge \\ & \qquad x_2 \neq mId \wedge x_2 \in membRes(bId) \wedge position(bId, mId) < position(bId, x_2))\}\end{aligned}$$

6.2.4 Detecting the Hypotheses

Each substitution S_{u_i} generated for decision path u_i is associated to a conjunction of hypotheses H_{u_i} under which S_{u_i} is valid. For each decision path, H_{u_i} is composed of the conditions of the form $q = w$, $\neg\ (q = w)$, $q \in S$ or $\neg\ (q \in S)$ labelling the path's edges and the conditions H_u generated for the first identification of the free variables of $\overrightarrow{k}$ (see Sect. 5.1).

6.2.4.1 General Process

There may be several input clauses in $IC_b(a)$ for the same attribute b. For each input clause $a^j(\overrightarrow{p}) : u^j$ of $IC_b(a)$, we have generated:

- $(\theta_u^j, S_u^j, H_u^j)$ if u^j is a functional term,
- (θ_u^j, H_u^j), and then $(\theta_{u_i}^j, S_{u_i}^j, H_{u_i}^j)$ for each decision path u_i^j, if u^j is a conditional term.

When all the u^j are functional terms, then the B substitution for state variable $b^\star$ is the following **IF THEN ELSE** expression:

IF $H_u^{1\star}$ THEN SH^1
ELSE IF $H_u^{2\star}$ THEN SH^2
...
ELSE IF $H_u^{j\star}$ THEN SH^j
...
END

where each condition $H_u^{j\star}$ is the B translation of condition H_u^j, and where each substitution SH^j is S_u^j.

If u^j is a conditional term, then the corresponding SH^j in the above-mentionned expression corresponds to a **IF THEN ELSE** substitution with conditions $H_{u_i}^{\star j}$ and substitutions $S_{u_i}^j$. Nevertheless, the conditions $H_{u_i}^j$ are not necessarily disjoined and the resulting substitution is then not correct in B. One can rewrite the expression by reusing the decision tree DT_b generated for attribute definition b. We consider in DT_b only the edges labelled by conditions of the form $q = w$, $q \in S$ or their negation, and we collect, for each node N of this new tree $DT_b^\#$ the list of substitutions $S_{u_i}^j$ with the same hypotheses H_{u^l}.

6.2.4.2 Algorithm for Conditional Terms

We build the new tree $DT_b^\#$ as follows. The root and the nodes with the edges labelled by the conditions H_u^j that correspond to step 1 in Sect. 5.2 are the same as in DT_b. Then, for each conditional term u^j, the corresponding subtree in DT_b is traversed in infix order to analyse the edges. We colour the edges of DT_b that correspond to a condition of the above-mentionned form and the corresponding children nodes. We build the new tree $DT_b^\#$ and we colour the traversed nodes and egdes of DT_b simultaneously.

Each node N of DT_b is associated to a node $N^\#$ of $DT_b^\#$. Let N be a node of the subtree. If N is a leaf, then we return to the father F of N to analyse the right child of F or the node F itself. If the left child N_l of N is labelled by a condition c of the form $q = w$ or $q \in S$, then a left child $N_l^\#$ labelled by condition c is created for $N^{\#}$ and N_l and its egde are coloured in DT_b. Node N_l is then analysed. Otherwise, if the left child N_l of N is not labelled by a condition of this form, then we analyse N_l and the corresponding node of N_l in $DT_b^{\#}$ is also $N^{\#}$.

Once the left child N_l of N is analysed, then the right child N_r is analysed. If the left child N_l of N is coloured, then the right child N_r is coloured as well, a right child $N_r^{\#}$ labelled by the same condition as N_r is created for $N^{\#}$ in $DT_b^{\#}$. Node N_r is then analysed. Otherwise, if N_l is not coloured, then the corresponding node of N_r is $N^{\#}$ and N_r is analysed.

Each node of $DT_b^{\#}$ is attached to a list of functional terms of DT_b. One way to verify that the tree is correctly built is to check that the conjunction of hypotheses H_{u^l} associated to each functional term u^l attached to node $N^{\#}$ in $DT_b^{\#}$ is the same as the conjunction of conditions that label the path's edges that lead to node N. The substitution for b is then a **IF THEN ELSE** expression generated as follows. Each **IF** condition corresponds to a condition that labels an edge of $DT_b^{\#}$. Let us recall that the tree is built such that a right child is labelled by the negation of the condition labelling the left child. A substitution corresponds to the application in their declaration order of the list of substitutions attached to a node in $DT_b^{\#}$. If two or more substitutions exist in this list for the same $\overrightarrow{k}$, then only the first one is taken into account. The result for a node M in $DT_b^{\#}$ is composed in parallel with the **IF THEN ELSE** expression that correspond to the subtree of M. The resulting expression is the B substitution for state variable $b^{\star}$.

6.2.4.3 Examples

A first example is the substitution for attribute definition $dueDate$ in operation $Transfer$. There are three decision paths that lead, respectively, to $CurrentDate + 365$, $CurrentDate + loanDuration(mId)$ and $\perp$. The latter corresponds to a $\lhd$ substitution. The conditions of the form $q = w$ or $q \in S$ are only specified at the end of the decision paths. By reusing the decision tree built for attribute $dueDate$, one can determine that the **IF THEN ELSE** expression only concerns the substitution on $dueDate$.

Sometimes, there are several substitutions attached to the same node in $DT_b^{\#}$. In that case, the substitution is the result of the application of the list of substitutions in their declaration order. For instance, there are two decision paths for input clauses Transfer in attribute $nbLoans$. The assignment values θ_1 and θ_2 for key attribute mId are $\{mId := q_2\}$ for the first decision path and $\{mId := borrower(q_1)\}$ for the second decision path. The substitutions are:

$$\begin{array}{l} nbLoans := nbLoans \lhd\!\!\!+ \{mId \mapsto nbLoans(mId) + 1\} \\ nbLoans := nbLoans \lhd\!\!\!+ \{borrower(bId) \mapsto nbLoans(borrower(bId)) - 1\} \end{array}$$

In this example, H_1 and H_2 are both $true$. Since there is only one input clause of Transfer for attribute $nbLoans$, the resulting substitution is:

$$\begin{array}{l} nbLoans := nbLoans \\ \quad \lhd\!\!\!+ \{mId \mapsto nbLoans(mId) + 1, borrower(bId) \mapsto nbLoans(borrower(bId)) - 1\} \end{array}$$

If the input clauses of $nbLoans$ for action Transfer were:

$$
\begin{array}{l}
\mathsf{Transfer}(bId, mId, \_) \ : nbLoans(mId) \ + \ 1 \\
\mathsf{Transfer}(bId, mId', \_) \ : \ \textbf{if } mId = borrower(bId) \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{then } nbLoans(mId) \ - \ 1 \textbf{ end}
\end{array}
$$

then, the generated substitutions for attribute *nbLoans* would be respectively:

$$
\begin{array}{l}
nbLoans := nbLoans \mathbin{\lhd\!\!\!+} \{mId \mapsto nbLoans(mId) + 1\} \\
nbLoans := nbLoans \mathbin{\lhd\!\!\!+} \{borrower(bId) \mapsto nbLoans(borrower(bId)) - 1\}
\end{array}
$$

These two substitutions would be applied in their declaration order and the final result would be the same as above-mentionned.

Chapter 7

Conclusion

There exist many integrated methods that combine state-based specifications with event-based specifications, like csp2B [But99] or CSP || B [ETLF04], but none of them is well adapted for the specification of information systems. Our aim is to use both B and EB^3 to specify such systems [Ger04].

Contrary to many other event-based languages, EB^3 extensively uses the notion of state since attribute definitions involve the valid traces of the system as well as common basic types to compute attribute values. This ability makes the translation into B easier. Attributes and, more generally, all the relevant data that the system manipulates need a state notion to be represented in a specification. As a matter of fact, information systems require state-based specifications for their data model and that is why this contribution is very important for our aims.

In this report, we have first shown how to generate substitution formulas and how to rewrite such formulas as usual B substitutions for EB^3 attribute definitions. We have then shown how to directly synthesize B substitutions from EB^3 attribute definitions. This is an important preliminary work for the synthesis of complete B specifications from correct EB^3 specifications. It allows us to:

- determine the minimal consistent B specification that the B model of an EB^3 specification must contain,
- define and verify static properties like data integrity constraints for IS by using the B model,
- implement imperative programs for relational databases.

7.1 Generating the Weakest Preconditions

The operations that we have presented in this report are not consistent, because the operation preconditions are only typing constraints on the input parameters. There are mainly two ways to make the synthesized operations consistent. One way is to analyse EB^3 process expressions and to generate adequate invariant properties and operation preconditions. This is a first work in progress, close to the CSP || B approach [TSB03]. Another way is to generate the weakest preconditions such that the operations become consistent. In this section, we briefly

Substitution	$[Substitution]R$
$skip$	R
$x := E$	$R[x \leftarrow E]$
PRE P **THEN** S **END**	$P \ \wedge \ [S]R$
IF C **THEN** S **ELSE** T **END**	$(C \Rightarrow [S]R) \ \wedge \ (\neg\, C \Rightarrow [T]R)$
SELECT Q **THEN** S **END**	$Q \Rightarrow [S]R$
ANY X **WHERE** Q **THEN** S **END**	$\forall\, X.\ (Q \Rightarrow [S]R)$
CHOICE S **OR** T **END**	$[S]R \ \wedge \ [T]R$
$S;T$	$[S][T]R$

Table 7.1: Semantics for the main operators of GSL

show how to generate the weakest preconditions such that these B operations become consistent.

7.1.1 General Process

The semantics of B substitutions is based upon the weakest precondition semantics of Dijkstra [Dij76]. Let S be a substitution and R a predicate, the following expression denotes the weakest precondition such that substitution S establishes predicate R:

$$[S]R$$

In other words, any precondition P stronger than $[S]R$ is correct for S. The generalized substitution language (GSL) of B is defined by this semantics (see Tab. 7.1 for the main operators). The semantics for each operator of the language is detailed in [Abr96]. For instance, substitution *skip* which denotes the substitution that does nothing is defined by:

$$[skip]R = R$$

If R is predicate *true*, then $[S]R$ is the termination predicate of substitution S.

Each operation of a B abstract machine must satisfy the following property:

$$I \wedge P \quad \Rightarrow \quad [S]I \tag{7.1}$$

where I is the invariant of the abstract machine, P is the operation precondition and S is the substitution in the body of the operation. The approach mentionned in Chapter 4 allows one to generate the invariant I of the machine and the substitutions S of each operation of the system. The generated preconditions TC only consist of typing constraints on the input parameters of operations. To satisfy the proof obligation (7.1), we define P as the conjunction of TC and $[S]I$. The main issue is therefore the computation of $[S]I$.

One way is to use the rules described in Tab. 7.1 to compute $[S]I$ for each operation of the system. Computations can be simplified or generalized, because the substitutions and the invariant predicates synthesized by the approach in Sect. 5 are always of the following forms. For substitutions, let V be a state variable, $e_1, \ldots, e_n$ be elements of a set state variable, and $v_1, \ldots, v_n$ be constants or input parameters:

- $V := V \cup \{v_1, \ldots, v_n\}$,
- $V := V \mathbin{\lhd\!\!\!+} \{e_1 \mapsto v_1, \ldots, e_n \mapsto v_n\}$,
- $V := V \; - \{e_1, \ldots, e_n\}$,
- $V := \{e_1, \ldots, e_n\} \lhd V$,
- or $V := V \rhd \{e_1, \ldots, e_n\}$.

For invariant predicates, let U, V, W be state variables and S be a given set:

- $V \subseteq S$,
- $V \in W \to S$,
- $V \in U \leftrightarrow W$,
- $V \in U \nrightarrow W$,
- $V \in U \to W$,
- $V \in U \rightarrowtail W$,
- $V \in U \twoheadrightarrow W$,
- $V \in U \rightarrowtail\!\!\!\!\!\twoheadrightarrow W$.

The above-mentionned semantics rules are used together with some rewriting rules that simplify expressions. We first consider the rules that involve surcharge ($\lhd\!\!\!+$) and antirestriction ($\lhd$). Functions can be simplified into partial functions with additionnal constraints. For instance, a total function $f : A \to B$ is equivalent to a partial function $f : A \nrightarrow B$ satisfying $dom(f) = A$. Consequently, simplification rules about *dom* and *ran* are also taken into account. The application of these rules for each substitution of each operation on each invariant predicate then gives us the weakest preconditions of the B operations.

Even if the substitutions and the invariant predicates are always of the same forms, the computation of weakest precondition remains a difficult task. Another way to compute them is the use of a logic solver. The best solution would be the creation of a special tool. However, some existing tools can be adapted or used differently from their primary goals in order to generate the weakest precondition of a substitution. For instance, by using Atelier B [Cle], we have been able to determine the main weakest preconditions of the operations from the B machine described in Chapters 5 and 6. It consists in specifying these operations with Atelier B and then in trying to prove the proof obligations (PO). Since the operation preconditions are only typing constraints, some subgoals of the POs cannot be proved. Most of them are the weakest preconditions such that operations become consistent. The other ones can be simplified with the above-mentionned rules in order to determine the weakest preconditions. This approach is similar to the identification of preconditions in Z with the tool Z/EVES described in [Led98].

7.1.2 Example

The process discussed in this section has been applied to the specification generated for the library management system (see Sect. 5). Operations **Acquire** and **Register** are already specified with the weakest preconditions. The other operations are modified as follows.

$$
\begin{array}{l}
\mathbf{Discard}(bId) \triangleq \\
\text{PRE } \ bId \in bookKey\_Set \ \wedge \ bId \notin dom(loan) \\
\text{THEN} \\
\quad bookKey := bookKey - \{bId\} \ || \\
\quad title := \{bId\} \lhd\!\!- title \\
\text{END;}
\end{array}
$$

$$
\begin{array}{l}
\mathbf{Unregister}(mId) \triangleq \\
\text{PRE } \ mId \in memberKey\_Set \ \wedge \ mId \notin ran(loan) \\
\text{THEN} \\
\quad memberKey := memberKey - \{mId\} \ || \\
\quad nbLoans := \{mId\} \lhd\!\!- nbLoans \\
\text{END;}
\end{array}
$$

$$
\begin{array}{l}
\mathbf{Lend}(bId, mId) \triangleq \\
\text{PRE } \ bId \in bookKey\_Set \ \wedge \ bId \in bookKey \ \wedge \ bId \notin dom(loan) \ \wedge \\
\qquad mId \in memberKey\_Set \ \wedge \ mId \in memberKey \\
\text{THEN} \\
\quad loan := loan \cup \{(bId, mId)\} \ || \\
\quad nbLoans := nbLoans \lhd\!\!+ \{mId \mapsto 1 + nbLoans(mId)\} \\
\text{END;}
\end{array}
$$

$$
\begin{array}{l}
\mathbf{Return}(bId) \triangleq \\
\text{PRE } \ bId \in bookKey\_Set \ \wedge \ nbLoans(borrower(bId)) \geq 1 \\
\text{THEN} \\
\quad loan := loan - \{(bId, borrower(bId))\} \ || \\
\quad nbLoans := nbLoans \lhd\!\!+ \{borrower(bId) \mapsto nbLoans(borrower(bId)) - 1\} \\
\text{END;}
\end{array}
$$

$$
\begin{array}{l}
\mathbf{Transfer}(bId, mId) \triangleq \\
\text{PRE } \ bId \in bookKey\_Set \ \wedge \ bId \in bookKey \ \wedge \\
\qquad nbLoans(borrower(bId)) \geq 1 \ \wedge \ bId \in dom(loan) \ \wedge \\
\qquad mId \in memberKey\_Set \ \wedge \\
\qquad mId \in memberKey \ \wedge \ mId \neq borrower(bId) \\
\text{THEN} \\
\quad loan := loan - \{(bId, borrower(bId))\} \cup \{(bId, mId)\} \ || \\
\quad nbLoans := nbLoans \lhd\!\!+ \{(mId \mapsto nbLoans(mId) + 1), \\
\qquad\qquad (borrower(bId) \mapsto nbLoans(borrower(bId)) - 1)\} \\
\text{END;}
\end{array}
$$

7.2 Determining and Verifying Data Integrity Constraints

The work described in this report is complementary with the verification of operation ordering that is described in [FL03]. By deriving B substitutions from EB^3 attribute definitions and by generating the weakest preconditions of operations, we are now able to define and verify new safety properties on the IS model.

In particular, data integrity constraints are properties that must always be satisfied by the IS data model. In EB^3, such properties are difficult to verify, because attributes are recursively defined on the valid traces of the system. Since attributes are represented by state variables in B, data integrity constraints then correspond to invariant properties. With the B translation of EB^3 attribute definitions, we can define new invariant properties for data integrity constraints and then generate the weakest preconditions such that operations remain consistent.

For instance, an integrity constraint for attribute *nbLoans* is the following: the number of loans of each member is limited to a certain limit, five books for example. Consequently, the following invariant must be satisfied at each execution of an operation:

$$\forall\, mId \in memberKey \bullet nbLoans(mId) \leq 5$$

This new invariant implies in particular that operation **Lend** must contain precondition $nbLoans(mId) \leq 4$ to satisfy the proof obligations. Such a precondition corresponds in EB^3 to a guard of action `Lend` in the process expression called `loanController` in Fig. 2.5. As a future work, we aim at verifying the consistency of EB^3 action guards by transposing them into the precondition of their corresponding B operation. We also want to determine new EB^3 action guards by defining new data integrity contraints specified as B invariant properties and by analysing their consequences on the weakest preconditions of operations.

7.3 Implementation

From the results presented in this paper, one can use B refinement techniques described in [Mam02] to implement the system with SQL. As a future work, we aim at directly synthesizing from EB^3 attribute definitions the required imperative programs for relational databases. We also want to implement tools to support the translations from EB^3 to B and from EB^3 to relational databases. To make the translation easier, we also want to define normal forms for input clauses of EB^3 attribute definitions.

Bibliography

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

[But99] M. Butler. csp2B : A practical approach to combining CSP and B. In J. Wing, J. Woodcock, and J. Davies, editors, *FM'99 World Congress on Formal Methods*, pages 223–241, Toulouse, France, 22-24th Sept 1999.

[Cle] Clearsy. Atelier B. `http://www.atelierb-societe.com`.

[Dij76] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

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

[ETLF04] N. Evans, H. Treharne, R. Laleau, and M. Frappier. How to verify dynamic properties of information systems. In *2nd IEEE International Conference SEFM*, Beijing, China, September 2004. IEEE Computer Society Press.

[FFL03] B. Fraikin, M. Frappier, and R. Laleau. A comparison of EB^3 and B for information system specification. In *State-oriented vs. Event-oriented thinking in Requirements Analysis, Formal Specification and Software Engineering (StEve)*, Pisa, Italy, September 13 2003. Satellite workshop at FM 2003.

[FL03] M. Frappier and R. Laleau. Proving event ordering properties for Information Systems. In *ZB2003: Formal Specification and Development in Z and B*, volume 2651 of *LNCS*, Turku, Finland, 4-6 June 2003. Springer-Verlag.

[FLN96] P. Facon, R. Laleau, and H.P. Nguyen. Mapping object diagrams into B specifications. In *Method Integration Workshop*, EWICS, Leeds, UK, 1996. Springer-Verlag.

[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.

[Ger04] F. Gervais. *EB^4 : Vers une méthode combinée de spécification formelle des systèmes d'information.* Dissertation for the general examination, GRIL, Université de Sherbrooke, Québec, June 2004.

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

[Led98] Y. Ledru. Identifying pre-conditions with the Z/EVES theorem prover. In *Proc. 13th International Conf. on Automated Software Engineering*. IEEE Computer Society Press, 1998.

[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.

[TSB03] H. Treharne, S. Schneider, and M. Bramble. Composing specifications using communication. In *ZB2003: Formal Specification and Development in Z and B*, volume 2651 of *LNCS*, Turku, Finland, 4-6 June 2003. Springer-Verlag.