Design of an Automatic Prover Dedicated to the Refinement of Database Applications

Amel Mammar and Régine Laleau

CEDRIC-IIE(CNAM) 18 allée Jean Rostand, 91025 Evry, France
{mammar, laleau}@iie.cnam.fr

Abstract. The paper presents an approach that enables the elaboration of an automatic prover dedicated to the refinement of database applications. The approach is based on a strategy of proof reuse and on the specific characteristics of such applications. The problem can be stated as follows. Having established a set of basic refinement proofs associated to a set of refinement rules, the issue is to study how these basic proofs can be reused to establish more elaborate refinements. Elaborate refinements denote refinements that require the application of more than one refinement rule. We consider the B refinement process. In B, substitutions are inductively built using constructors. For each B constructor, we have formally defined the necessary and sufficient conditions that enable the reuse of the basic proofs. An application of our approach to data-intensive applications is presented.

Keywords

Refinement process, Proof reuse, B method, Data-intensive applications

1 Introduction

The last decade has seen a growing use of databases in several different domains: e-business, financial systems, smart cards, etc. Although these areas are not critical (no human risk), economic interests are involved and a certain degree of safety is required. Our project aims at providing users with a complete formal environment for the specification and the development of database applications [13]. For this, we use the B formal method, developed by Jean-Raymond Abrial [1]. It is a complete method that supports a large segment of the development life cycle: specification, refinement and implementation. It ensures, thanks to refinement steps and proofs, that the code satisfies its specification. It has been used in significant industrial projects and commercial case tools [2, 3] are available in order to help the specifier during the development process.

The specification of a database application is composed of two parts: specification of the data structure by using an Entity/Relationship model and specification of user transactions which describe the functionalities of the system. These transactions are built on a set of generic basic operations (insert, delete or update elements). We have proposed a method that allows this specification to be described using the B specification language [9]. The obtained specification is then refined up to an implementation using the relational database model. In the database area, data refinement is usually achieved by the application of a well-known algorithm that generates a relational schema from an Entity/Relationship schema [8, 6]. In the B method, refinement process is generally manual, since target implementation languages can be various. We have defined a specific refinement process, dedicated to our application domain, that is automated [17]. A set of elementary rules refining the data structure and the basic operations have been elaborated [10]. It is based on the above-mentioned algorithm. An elementary refinement proof is associated to each rule, that ensures the correctness of the transformation.

Software engineers are resistant to use approaches based on formal methods mainly because of the proof phase that requires significant skills. In order to assist them, we have considered the automation of the refinement proof. In other words, we have studied the automation of transaction refinement. The problem addressed in the paper can be stated in this way. The B refinement process being monotonous, the refinement of a transaction comes down to the refinement of the basic operations which it is built on and the refinement of the B constructors used to combine the basic operations. Thus, is it possible to reuse the elementary refinement proofs to establish the proof of the refinement of a transaction ? For each B constructor, we have defined a set of reuse constraints, independently of any application domain. Then we have demonstrated that, in the database area, most of these constraints are always satisfied. This allows a great number of refinement proofs to be automatically discharged. This is a very interesting result. Indeed, if we want to extend the use of formal methods in other domains than those where they are usually applied (i.e. critical systems) we absolutely need to provide assistant tools with a lot of things which are automatically achieved or produced.

Improvement of proof processes by reusing already computed proofs is an active research area. Several techniques have been developed. Most of them are developed for the proof by induction of mathematical theorems. Among them, we can mention: *reusing by transformation* [15, 16], *reusing by type isomorphism* [5], *reusing by generalization* [18, 19] and *reusing by sub-typing* [14]. In [16], two different representations of natural numbers are considered: a binary representation and the usual representation using the two constructors 0 and successor. The authors have developed a tool that transforms each proof computed within one of the two representations into the other. In [14], if the type A is a subtype of B, a coercion function that transforms each term of type A into a term of type B is defined. This function permits to replay for A all the proofs already computed for B. This technique is largely used within the *Coq* prover which is a strongly typed language [4]. In [5], a theoretical foundation for proof reuse, based on type isomorphisms in dependent type theory is presented. Our work can be compared with the works of [18] and [19]. These works are based on the analysis of already established proofs, by producing an explanation or a justification of why it is successful. In order to reuse it, both the formula and its proof are generalized. To our knowledge, reuse of proofs in a refinement proof process remains an unexplored problem.

In the following, we briefly give an overview of the B refinement process (Section 2). In section 3, we present the framework of our work and explain why we can consider to define a reuse proof strategy in the refinement process of data-intensive applications. A formal definition of reuse constraints is presented in Section 4. Section 5 describes how they behave for database applications and enable the implementation of an automatic prover within Atelier B. The benefits expected from such reuse strategy and future works are discussed in Section 6.

2 Overview of B and its refinement process

The B language is based on first order logic extended to set constructors and relations. The operations are specified in the generalized substitution language, which is a generalization of the Dijkstra's guarded command notations. The B method is a model-based method. A system is described in terms of abstract machines that contain state variables, invariant properties expressed on the variables and operations described in terms of preconditions and substitutions.

Refinement is the process that transforms an abstract specification into a less abstract one. These transformations operate on data and/or operations. Data are refined by adding new variables or replacing the existing ones by others which are supposed to be more concrete (closer to a target implementation language). Operation refinement consists in eliminating non-determinism. The last step of the refinement process produces an implementation component which is to be used as the basis for translation into executable code. Both specification and refinement give rise to proof obligations. Specification proofs ensure that operations preserve the invariant, whereas refinement proofs ensure the correctness of a refined component with respect to its initial component.

In order to prove the correctness of refinements, we use the relational semantics of substitutions based on the definition of the two predicates $Trm(S)$ and $Prd(S)$ associated to any substitution S. These predicates are defined in the B-Book [1] by:

$$Trm(S) \underline{\underline{\Delta}} [S](x = x) \ . \tag{1}$$

$$Prd_{x,x'}(S) \underline{\underline{\Delta}} \neg [S](x' \neq x) \ . \tag{2}$$

Intuitively:
$Trm(S)$: gives the necessary and sufficient condition for the termination of S,
$Prd_{x,x'}(S)$: gives the link between the values of variables x before (denoted x) and after (denoted x') the execution of S.

Notations: In the remainder of the paper, we need the following notations:

- Each substitution S is indexed by the set x of all the variables of the specification where S is defined : S_x
- $Prd_{y,y'}(S_x)$ where $y \subseteq x$ means that we consider the restriction of $Prd_{x,x'}(S_x)$ to the set of variables y. In general, y represents the set of the variables modified by S_x.

– In order to simplify expressions, $Prd_{x,x'}(S_x)$ is denoted $Prd(S_x)$.

Correctness of refinements: With the previous definitions, correctness of refinements is expressed as follows.
Let S_a and T_b be two substitutions. Let $J(a,b)$ be the predicate, called the gluing invariant, that states the relation existing between a and b. T_b refines S_a according to $J(a,b)$, denoted $S_a \sqsubseteq_{J(a,b)} T_b$, iff:

$$\exists\, u, v.\ J(u,v) \tag{3}$$

$$\forall\, a, b \cdot (Trm(S_a) \wedge J(a,b)) \rightarrow$$
$$[Trm(T_b) \wedge \forall\, b' \cdot (Prd(T_b) \rightarrow \exists\, a' \cdot (Prd(S_a) \wedge J(a', b')))] \tag{4}$$

Let us explain each condition:
(3) means that the gluing invariant must be satisfiable(it isn't a contradiction),
(4) means firstly that for each possible interpretation of a that ensures the termination of S_a, the corresponding set b (that satisfies the gluing invariant) ensures the execution of T_b, and secondly that, for each possible result b' of T_b, there must exist a corresponding result a' of S_a such that a' and b' satisfy the gluing invariant J.

Simplification of refinement proofs: For some types of substitutions (e.g. assignment), the proof of its termination doesn't raise any difficulty : $Trm(T_b)$ is trivially true. Thus the proof of correctness of a refinement consists in exhibiting a value of a', associated to a given value b' satisfying $Prd(T_b)$, that satisfies the two predicates $Prd(S_a)$ and $J(a', b')$. So we have to prove that:

$$\exists\, a' \cdot (Prd(S_a) \wedge J(a', b'))$$

for given values of a, b, b' satisfying:

$$Trm(S_a) \wedge J(a,b) \wedge Prd(T_b)$$

We will use this simplified proof obligation to prove refinements in Section 3 because $Trm(T_b)$ is always trivially true in the examples we are considering.

3 Reuse of proofs during the refinement process: an illustrative example

Our project aims at providing software engineers with a formal method for the specification and the development of safety data-intensive applications. The global framework of our project, represented by Figure 1, includes two main phases: *Specification* and *Refinement*. Both phases use a formal method (B in our case) and thus can be validated by proofs. Nevertheless, establishing proofs, and specially refinement proofs, is a long, hard and tedious task. A solution to assist engineers is to build tools in order to automate parts of the proofs. Of course, this is not possible in general but only for specific areas where a generic strategy of refinement may be defined. This is what we propose for data-intensive applications.

Firstly, we describe the characteristics of the abstract specifications of such applications. Then we illustrate, through an example, the refinement process and how it can be automated.

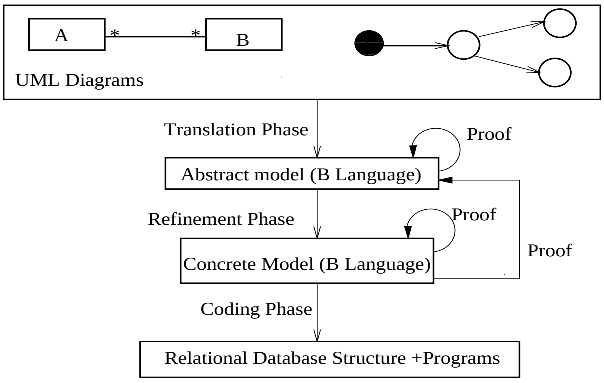

Fig. 1. Formal mapping of a UML conceptual model into a relational implementation using the B method

3.1 Elaboration of the abstract model of an application

The first step of our development is to construct a B abstract model. It is derived from UML diagrams (class diagram, state/ collaboration diagrams) by using a set of translation rules. A description of this translation and a comparison with other work are already described in [9, 11] and are out of the scope of this paper. Which is important to have in mind in order to understand our approach is that the different UML diagrams are dedicated to data-intensive applications, with precise semantics defined in [12]. In particular, a class diagram is defined with the semantics of an E/R diagram. A consequence is that the specification we generate has always the same characteristics. It is composed of two layers:

a)The internal layer contains all the variables that define the state of the application and a set of basic operations that act upon them. The variables represent the classes, associations and attributes of the class diagram. The basic operations comprise insert and delete operations for each class and association and update operations that change the value of the attributes.

Let us consider the example of a simplified video club. A cassette is either loaned by a customer or available in a shop. Each customer and each shop are identified by an attribute, called a key, respectively *NumCu* and *NumSh*. They are natural numbers. Figure 2 presents the corresponding UML class diagram which gives a synthetic view of the data of the application.

The derived B specifications are as follows (just the relevant parts are given):

$$Loan \in Cassette \nrightarrow Customer \wedge Available \in Cassette \nrightarrow Shop \wedge$$
$$NumCu \in Customer \rightarrowtail NAT \wedge NumSh \in Shop \rightarrowtail NAT$$

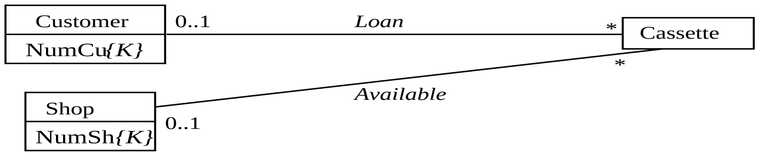

Fig. 2. Class diagram of a video club

Cassette, *Customer* and *Shop* are variables representing the existing instances of the corresponding classes; the *Loan* and *Available* variables represent the corresponding monovalued[1] associations.

Basic operations are automatically generated from the class diagram. For example, $AddLoan(ca, cu)$ creates a new link in the association *Loan* that relates a cassette ca to a customer cu; $DeleteAvailable(\{ca\})$ deletes the links, related to a given set of cassette (here $\{ca\}$)), from the association *Available*. In B, these operations are expressed by the two substitutions [2]:

$$Loan := Loan \cup \{ca \mapsto cu\} \text{ and } Available := \{ca\} \lhd Available$$

b) In the external layer, user transactions are specified. A transaction is specified by a B operation that calls the basic operations of the internal layer using different B constructors (test, parallel substitution, ...). No additional variables are defined in this layer. Moreover, variables of the internal level can be modified only by using basic operations. For example, let us consider the transaction $LoanCassette(ca, cu, sh)$ that loans a cassette ca to a given customer cu from a shop sh. This transaction is constructed by calling the two basic operations *DeleteAvailable* and *AddLoan* using *Parallel* and *IF* constructors as follows[3]:

IF $Available(ca) = sh$ **THEN**
    $AddLoan(ca, cu) \parallel$
    $DeleteAvailable(\{ca\})$
END

Note that in reality a class diagram may contain a great number of classes and associations, whereas a transaction involves few classes and associations and has a low algorithmic complexity. The transaction *LoanCassette* has been defined in order to facilitate an intuitive presentation of our approach. A more realistic transaction would be a little bit more complex, which would just require more steps of refinement.

1 An association is called monovalued if one of its maximum multiplicities is 1

2 The complete operations include preconditions that we omit here because they haven't any influence on our refinement process

3 Such a transaction can be also described using state/collaboration diagrams, more detail can be found in [9, 11].

3.2 The refinement process

The second phase of our approach consists in refining the B abstract specification obtained in the previous phase in order to generate a relational database implementation. Taking into account the specific characteristics of the specification, the global refinement can be achieved in two successive steps: the first step concerns the refinement of the internal layer (i.e. variables and basic operations), the second one concerns the refinement of the external layer (i.e. user transactions).

a) Variables and basic operations refinement
The refinement is based on the algorithm used in database design [6], thus we exactly know how the data are refined. This allows us to define a set of elementary generic refinement rules that act both on variables and basic operations [10].

Let us take the example of the refinement rules related to a monovalued asociation. Let C and D be two classes linked by a monovalued association f. In B, f is specified by a function from the variable C to the variable D. Let v be the key of the class D, of type T. In B, v is specified by an injective total function from D to T. The data refinement of the association f consists in replacing f by a new attribute, called f_1, in class C which is a reference to the key of D (in a relational database, it will be defined by a foreign key). In B, f_1 is specified by a function from C to T and the gluing invariant is $f_1 = (v \circ f)$. The refinement of a basic operation related to an association is just a rewriting of the operation in order to take into account the association refinement. The following table sums up the refinement of the *add* and *delete* operations, which *AddLoan* and *DeleteAvailable* are instantiation of:

RULE	Abstract_Subst.	Concrete_Subst.	Gluing invariant
Rule_Add	$f := f \cup \{c \mapsto d\}$	$f_1 := f_1 \cup \{c \mapsto v(d)\}$	$f_1 = (v \circ f)$
Rule_Del	$f := C_1 \lhd f$	$f_1 := C_1 \lhd f_1$	$f_1 = (v \circ f)$

where:
– c and d are elements of C and D respectively.
– C_1 is a subset of C.

To establish the correctness of *Rule_Add* and *Rule_Del*, we have to carried out the two following proofs:

- Proof of *Rule_Add* (P_{Add}): $\exists f'.(Prd(f := f \cup \{c \mapsto d\}) \wedge (f_1' = v \circ f'))$ for values of f, v, f_1, f_1' satisfying: $(f_1 = v \circ f) \wedge Prd(f_1 := f_1 \cup \{c \mapsto v(d)\})$

- Proof of *Rule_Del* (P_{Del}): $\exists f'.(Prd(f := C_1 \lhd f) \wedge (f_1' = v \circ f'))$ for values of f, v, f_1, f_1' satisfying: $(f_1 = (v \circ f) \wedge Prd(f_1 := C_1 \lhd f_1))$

The proofs of these two rules are discussed in Section 5.

We have defined about 120 rules. With Atelier B (version 3.5), 70% of the proofs of these rules have been automatically discharged. However, just the easier proofs are concerned: the proofs related to the *Trm* predicates. The remaining proofs were rather hard and sometimes very tedious to achieve. Nevertheless, due to the generic feature of the rules, it is possible to define proof tactics that enable the automation of the refinement proofs. It means that, once the proof of a generic refinement rule has been achieved, it is possible to reuse it in all the instantiations of the rule.

Let us apply this strategy on our example. *AddLoan* and *DeleteAvailable* are refined by the refinement rules *Rule_Add* and *Rule_Del*. This gives the two concrete substitutions:

$$Available_1 := \{ca\} \lhd Available_1 \text{ and } Loan_1 := Loan_1 \cup \{ca \mapsto NumCu(cu)\}$$

To establish the correctness of these refinements, we just need to instantiate the generic proofs (P_{Add}) and (P_{Del}). This instantiation is achieved by taking: ($f = Loan, v = NumCu, c = ca, d = cu$) and ($f = Available, v = NumSh, C_1 = \{ca\}$) respectively.

b)Transactions refinement

The basic operations being refined, the following step deals with the refinement of user transactions. Recall that transactions act only upon variables defined in the internal layer. Thus the refinement of a transaction is an algorithmic refinement. As the specification of a transaction is based on the specification of basic operations, the refinement of a transaction uses the refinement of these latter. More precisely, the B refinement process being monotonous, the refinement of each transaction comes down to the refinement of the operations that it calls and the refinement (or rewriting) of the different B constructors that relate these operation calls.

For example, let us take the refinement of the transaction *LoanCassette*. The refinement step that corresponds to the refinement of the associations *Loan* and *Available* reuses the refinement of the basic operations *AddLoan* and *DelAvailable*. In addition, the predicate of the *IF* constructor is rewritten according to the concrete variables $Loan_1$ and $Available_1$. So, the transaction *LoanCassette* is refined by [4, 5]:

$$\begin{array}{l}
\textbf{IF } (NumSh^{-1} \circ Available_1)(ca) = sh \textbf{ THEN} \\
\qquad Available_1 := \{ca\} \lhd Available_1 \parallel \\
\qquad Loan_1 := Loan_1 \cup \{ca \mapsto NumCu(cu)\} \\
\textbf{END}
\end{array}$$

As the variables *Loan* and *Available* are refined separately, the gluing invariant is equal to the conjunction of the gluing invariants associated to the refinement of the associations *Available* and *Loan*: $(Available_1 = NumSh \circ Available) \wedge (Loan_1 = NumCu \circ Loan)$.

To simplify expressions, we use the following notations:

[4] The parallel constructor will be refined by the sequence constructor during one of the next steps

[5] The exact specification would be two calls to the operations that correspond to the refinement of *AddLoan* and *DelAvailable*.

- $S_1 \mathrel{\widehat{=}} (Available := \{ca\} \lhd Available)$ and $S_2 \mathrel{\widehat{=}} (Loan := Loan \cup \{ca \mapsto cu\})$
- $T_1 \mathrel{\widehat{=}} (Available_1 := \{ca\} \lhd Available_1)$ and $T_2 \mathrel{\widehat{=}} (Loan_1 := Loan_1 \cup \{ca \mapsto NumCu(cu)\})$
- $a_1 \mathrel{\widehat{=}} \{Available\}$ and $a_2 \mathrel{\widehat{=}} \{Loan\}$
- $b_1 \mathrel{\widehat{=}} \{Available_1\}$ and $b_2 \mathrel{\widehat{=}} \{Loan_1\}$
- $a = a_1 \cup a_2$ and $b = b_1 \cup b_2$
- $J(a,b) \mathrel{\widehat{=}} (J(a_1,b_1) \wedge J(a_2,b_2)) \mathrel{\widehat{=}} (Available_1 = NumSh \circ Available) \wedge (Loan_1 = NumCu \circ Loan)$
- $C_1 \mathrel{\widehat{=}} (Available(ca) = sh)$ and $C_2 \mathrel{\widehat{=}} (NumSh^{-1} \circ Available_1(ca) = sh)$

To check the correctness of the refinement, we must prove:

$$\exists\, a'.[Prd(\mathbf{IF}\, C_1\, \mathbf{THEN}(S_1 \| S_2)\mathbf{END}) \wedge J(a',b')] \qquad (\mathrm{G}_1)$$

for given values of a, b, b' satisfying the following hypotheses:

$$\begin{aligned} &Trm(\mathbf{IF}\, C_1\, \mathbf{THEN}(S_1 \| S_2)\mathbf{END}) \wedge \\ &Prd(\mathbf{IF}\, C_2\, \mathbf{THEN}(T_1 \| T_2)\mathbf{END}) \wedge \\ &J(a,b) \end{aligned}$$

Not surprisingly, this proof is not automatically discharged by Atelier B. However, at a certain step of the elaboration of the interactive proof, the prover comes down to the proofs (P_{Add}) and (P_{Del}) (instantiated on *AddLoan* and *DelAvailable*). It is very interesting because it means that it is possible to reuse the basic proofs established for the refinement of the basic operations. In the following, we explain how the interactive proof is elaborated.

By applying the definition of *Prd* and *Trm* of a IF-THEN-ELSE substitution, G_1 becomes:

$$\exists\, a'.[(C_1 \rightarrow Prd(S_1 \| S_2)) \wedge (\neg C_1 \rightarrow (a'_1 = a_1 \wedge a'_2 = a_2)) \wedge J(a',b')] \qquad (\mathrm{G}_2)$$

for given values of a, b, b' satisfying the following hypotheses:

$$\begin{aligned} &True \wedge \\ &(C_2 \rightarrow Prd(T_1 \| T_2) \wedge (\neg C_2 \rightarrow (b'_1 = b_1 \wedge b'_2 = b_2))) \wedge \\ &J(a,b) \end{aligned}$$

To prove the goal G_2, the prover generates four sub-proofs given in the following table:

	Goal	**Hypotheses**
(1)	$\exists\, a'.(Prd(S_1 \| S_2) \wedge J(a',b'))$	$C_1 \wedge C_2 \wedge Prd(T_1 \| T_2) \wedge J(a,b)$
(2)	$\exists\, a'.(Prd(S_1 \| S_2) \wedge J(a',b'))$	$C_1 \wedge \neg C_2 \wedge b'_1 = b_1 \wedge b'_2 = b_2 \wedge J(a,b)$
(3)	$\exists\, a'.(a'_1 = a_1 \wedge a'_2 = a_2 \wedge J(a',b'))$	$\neg C_1 \wedge C_2 \wedge Prd(T_1 \| T_2) \wedge J(a,b)$
(4)	$\exists\, a'.(a'_1 = a_1 \wedge a'_2 = a_2 \wedge J(a',b'))$	$\neg C_1 \wedge \neg C_2 \wedge b'_1 = b_1 \wedge b'_2 = b_2 \wedge J(a,b)$

The last proof is automatically discharged by the prover. Indeed, it is trivially true for $a'_1 = a_1$ and $a'_2 = a_2$ because of the hypotheses $b'_1 = b_1$, $b'_2 = b_2$ and the

invariant $J(a,b)$. The second and the third are proved interactively by reductio ad absurdum. Indeed the predicate C_2 being a rewriting of the predicate C_1, the two predicates are equivalent. So, both the conjunctions $(C_1 \wedge \neg C_2)$ and $(\neg C_1 \wedge C_2)$ are contradictory. It remains to interactively achieve the first proof (1).
As the set of variables modified by S_1 (resp. S_2) and the set of variables not modified by S_2 (resp. S_1) are disjoint, $Prd(S_1 \| S_2)$ is rewritten into $Prd(S_1) \wedge Prd(S_2)$. For the same reasons, the hypothesis $Prd(T_1 \| T_2)$ is rewritten into $Prd(T_1) \wedge Prd(T_2)$. Moreover, the sets of variables a_1 and a_2 are disjoint. So, the goal and the hypotheses of the proof (1) are rewritten into:

$$\exists\, a_1'.(Prd(S_1) \wedge J(a_1', b_1')) \tag{G$_3$}$$

under the hypotheses: $Prd(T1) \wedge J(a_1, b_1) \wedge C_1 \wedge C_2$, and

$$\exists\, a_2'.(Prd(S_2) \wedge J(a_2', b_2')) \tag{G$_4$}$$

under the hypotheses: $Prd(T2) \wedge J(a_2, b_2) \wedge C_1 \wedge C_2$.

Note that the hypotheses C_1 and C_2 are not relevant to prove G_3 and G_4. It is easy to remark that these two proofs are exactly the same that have been already established when we have proved the correctness of the basic operations *AddLoan* and *DeleteAvailable* refinements. Thus, they are discharged.

This example shows that *under some conditions* (above, conditions on sets of variables), it is possible to reuse the proofs of the refinement of the basic operations which a transaction is constructed on, in order to prove the refinement of the transaction. The different case studies we have carried out have confirmed this fact: most of the proofs to be achieved for proving the refinement of a transaction come down to the proofs of the refinement of its basic operations, whatever the B constructors used to combine these basic operations. The other proofs concern the refinement of the constructors themselves and are generally automatically discharged. This has led us to examine in more detail the formal definition of conditions that make the reuse of elementary refinement proofs possible. The objective of such a study is to provide a formal basis to the development of an automatic prover.

4 Defining reuse constraints

In B, substitutions are inductively constructed using B constructors (precondition, parallel, sequence, ...). For each B constructor, we have defined the necessary and sufficient conditions for reusing proofs. The complete results are detailed in [17]. Hereafter, we present the case of the parallel constructor.

Assume that we have already computed the proofs for the two elementary refinement rules:

$$S_{a_1} \sqsubseteq_{J_{ST}(a_1,a_2)} T_{a_2} \tag{H$_1$}$$

$$U_{b_1} \sqsubseteq_{J_{UV}(b_1,b_2)} V_{b_2} \tag{H$_2$}$$

Let $J(a_1 \cup b_1, a_2 \cup b_2)$ be a new predicate such that:

$$(S_{a_1} \| U_{b_1}) \sqsubseteq_{J(a_1 \cup b_1, a_2 \cup b_2)} (T_{a_2} \| V_{b_2}) \tag{G$_5$}$$

By definition of refinement, (H_1) and (H_2) give the following hypotheses:

$$\begin{array}{r} \forall\, a_1, a_2.(\mathit{Trm}(S_{a_1}) \wedge J_{ST}(a_1, a_2)) \rightarrow \\ [\mathit{Trm}(T_{a_2}) \wedge \forall\, a_2'.(\mathit{Prd}(T_{a_2}) \rightarrow \exists\, a_1'.(\mathit{Prd}(S_{a_1}) \wedge J_{ST}(a_1', a_2')))] \end{array} \tag{H$_3$}$$

$$\begin{array}{r} \forall\, b_1, b_2.(\mathit{Trm}(U_{b_1}) \wedge J_{UV}(b_1, b_2)) \rightarrow \\ [\mathit{Trm}(V_{b_2}) \wedge \forall\, b_2'.(\mathit{Prd}(V_{b_2}) \rightarrow \exists\, b_1'.(\mathit{Prd}(U_{b_1}) \wedge J_{UV}(b_1', b_2')))] \end{array} \tag{H$_4$}$$

And the goal (G_5) gives:

$$\begin{array}{l} \forall\, c_1, c_2.(\mathit{Trm}(S_{a_1} \| U_{b_1}) \wedge J(c_1, c_2)) \rightarrow [\mathit{Trm}(T_{a_2} \| V_{b_2}) \wedge \\ \quad \forall\, c_2'.(\mathit{Prd}(T_{a_2} \| V_{b_2}) \rightarrow \exists\, c_1'.(\mathit{Prd}(S_{a_1} \| U_{b_1}) \wedge J(c_1', c_2')))] \end{array} \tag{G$_6$}$$

where c_1 (resp. c_2) denotes the union of a_1 and b_1 (resp a_2 and b_2).

The aim of the section is to present the reasoning that leads to, on one hand, the determination of the conditions under that the already computed proofs (H_3) and (H_4) can be reused to prove (G_6) and, on the other hand, the actual reuse of these proofs.

4.1 A relevant rewriting of the goal (G_6)

The first issue is to rewrite (G_6) in order to exhibit parts of (H_3) and (H_4). In order to express the different predicates, we partition each set of variables (a_i and b_i) into the set of the modified variables (a_i^m and b_i^m) and the set of the unchanged ones (a_i^f and b_i^f). Using these new variables, the four predicates $\mathit{Trm}(S_{a_1} \| U_{b_1})$, $\mathit{Trm}(T_{a_2} \| V_{b_2})$, $\mathit{Prd}(S_{a_1} \| U_{b_1})$ and $\mathit{Prd}(T_{a_2} \| V_{b_2})$ are defined as follows[6]:

$$\begin{aligned} \mathit{Trm}(S_{a_1} \| U_{b_1}) &= \mathit{Trm}(S_{a_1}) \wedge \mathit{Trm}(U_{b_1}) \\ \mathit{Trm}(T_{a_2} \| V_{b_2}) &= \mathit{Trm}(T_{a_2}) \wedge \mathit{Trm}(V_{b_2}) \\ \mathit{Prd}(S_{a_1} \| U_{b_1}) &= \mathit{Prd}_{a_1^m, a_1^{m\prime}}(S_{a_1}) \wedge \mathit{Prd}_{b_1^m, b_1^{m\prime}}(U_{b_1}) \wedge d_1' = d_1 \\ \mathit{Prd}(T_{a_2} \| V_{b_2}) &= \mathit{Prd}_{a_2^m, a_2^{m\prime}}(T_{a_2}) \wedge \mathit{Prd}_{b_2^m, b_2^{m\prime}}(V_{b_2}) \wedge d_2' = d_2 \end{aligned}$$

where d_1 (resp. d_2) denotes the sub-set of variables of $a_1 \cup b_1$ (resp. $a_2 \cup b_2$) that the substitution $S_{a_1} \| U_{b_1}$ (resp. $T_{a_2} \| V_{b_2}$) doesn't modify. So, we have:

$$d_i = (a_i^f \cup b_i^f) - (a_i^m \cup b_i^m) \tag{H$_5$}$$

By substituting these definitions in (G_6), eliminating of the universal quantifier on the variables c_1 and c_2, and applying the deduction theorem, we obtain the following three goals:

[6] $d_i' = d_i$ denotes a conjunction of a set of equalities. Each equality, of the form $x' = x$, is related to one variable x of d_i.

$$Trm(T_{a_2}) \quad (\mathrm{G}_7)$$
$$Trm(V_{b_2}) \quad (\mathrm{G}_8)$$
$$\forall c_2'.(Prd_{a_2^m, a_2^{m\prime}}(T_{a_2}) \wedge Prd_{b_2^m, b_2^{m\prime}}(V_{b_2}) \wedge d_2' = d_2$$
$$\rightarrow \exists c_1{}'.(Prd_{a_1^m, a_1^{m\prime}}(S_{a_1}) \wedge Prd_{b_1^m, b_1^{m\prime}}(U_{b_1}) \wedge d_1' = d_1 \wedge J(c_1', c_2'))) \quad (\mathrm{G}_9)$$

under the additional hypothesis:

$$Trm(S_{a_1}) \quad (\mathrm{H}_6)$$
$$Trm(U_{b_1}) \quad (\mathrm{H}_7)$$
$$J(c_1, c_2) \quad (\mathrm{H}_8)$$

4.2 Identification of the reuse conditions

Proof of ($\mathbf{G}_7$) by reuse. In order to prove (G_7) by reusing (H_3), we must be able to deduce from our current environment the two hypotheses $Trm(S_{a_1})$ and $J_{ST}(a_1, a_2)$. $Trm(S_{a_1})$ being the hypothesis (H_6), it remains to derive from the hypothesis (H_6), (H_7) and (H_8), the predicate $J_{ST}(a_1, a_2)$. So, the first condition of reuse is:

$$\boxed{(Trm(S_{a_1}) \wedge Trm(U_{b_1}) \wedge J(c_1, c_2)) \rightarrow J_{ST}(a_1, a_2)} \quad (\mathrm{H}_9)$$

(H_6), (H_7), (H_8), (H_9) and the instantiation of the variables a_1 and a_2 by themselves in (H_3) prove therefore (G_7) and give an additional hypothesis:

$$\forall a_2{}'.(Prd(T_{a_2}) \rightarrow \exists a_1'.(Prd(S_{a_1}) \wedge J_{ST}(a_1{}', a_2{}'))) \quad (\mathrm{H}_{10})$$

Proof of ($\mathbf{G}_8$) by reuse. This proof is similar to the previous one. It requires the following condition:

$$\boxed{(Trm(S_{a_1}) \wedge Trm(U_{b_1}) \wedge J(c_1, c_2)) \rightarrow J_{UV}(b_1, b_2)}$$

and gives the following new hypothesis:

$$\forall b_2{}'.(Prd(V_{b_2}) \rightarrow \exists b_1'.(Prd(U_{b_1}) \wedge J_{UV}(b_1', b_2{}')) \quad (\mathrm{H}_{11})$$

Proof of ($\mathbf{G}_9$) by reuse. By eliminating of the universal quantifier and applying of the deduction theorem, the goal becomes :

$$\exists c_1{}'.(Prd_{a_1^m, a_1^{m\prime}}(S_{a_1}) \wedge Prd_{b_1^m, b_1^{m\prime}}(U_{b_1}) \wedge d_1' = d_1 \wedge J(c_1', c_2')) \quad (\mathrm{G}_{9.1})$$

under the additional hypothesis :

$$Prd_{a_2^m, a_2^{m\prime}}(T_{a_2}) \quad (\mathrm{H}_{12})$$
$$Prd_{b_2^m, b_2^{m\prime}}(V_{b_2}) \quad (\mathrm{H}_{13})$$
$$d_2' = d_2 \quad (\mathrm{H}_{14})$$

The instantiation of the universally quantified variables $a_2{}'$ and $b_2{}'$ by themselves in (H_{10}) and (H_{11}) and the substitution of a_2 and b_2 by $(a_2^m \cup a_2^f)$ and $(b_2^m \cup b_2^f)$ respectively give :

$$(Prd_{a_2^m, a_2^{m\prime}}(T_{a_2}) \wedge a_2^{f\prime} = a_2^f) \rightarrow \exists\, a_1'.(Prd(S_{a_1}) \wedge J_{ST}(a_1', a_2^{m\prime} \cup a_2^{f\prime})) \quad (\mathrm{H}_{15})$$

$$(Prd_{b_2^m, b_2^{m\prime}}(V_{b_2}) \wedge b_2^{f\prime} = b_2^f) \rightarrow \exists\, b_1'.(Prd(U_{b_1}) \wedge J_{UV}(b_1', b_2^{m\prime} \cup b_2^{f\prime})) \quad (\mathrm{H}_{16})$$

In order to be able to reuse these hypotheses to prove $(\mathrm{G}_{9.1})$, the two hypotheses:

$$a_2^{f\prime} = a_2^f \quad (\mathrm{H}_{17})$$

$$b_2^{f\prime} = b_2^f \quad (\mathrm{H}_{18})$$

must appear in (H_{14}). This means that each unchanged variable of T_{a_2} (resp. V_{b_2}) must remain unchanged by $(T_{a_2} \| V_{b_2})$. So:

$$\boxed{a_2^f \cap b_2^m = \emptyset \qquad \mathit{and} \qquad b_2^f \cap a_2^m = \emptyset}$$

(H_{12}), (H_{17}) and (H_{15}); (H_{13}), (H_{18}) and (H_{16}) give:

$$\exists\, a_1'.(Prd_{a_1, a_1'}(S_{a_1}) \wedge J_{ST}(a_1', a_2^{m\prime} \cup a_2^f)) \quad (\mathrm{H}_{19})$$

$$\exists\, b_1'.(Prd_{b_1, b_1'}(U_{b_1}) \wedge J_{UV}(b_1', b_2^{m\prime} \cup b_2^f)) \quad (\mathrm{H}_{20})$$

4.3 Actual reuse of the proofs

Let $sol_{a_1'}$ and $sol_{b_1'}$ be one of the already computed values of a_1' and b_1' that satisfy the hypotheses (H_{19}) and (H_{20}) respectively. So:

$$[a_1' := sol_{a_1'}]^7 Prd_{a_1, a_1'}(S_{a_1}) \wedge [a_1' := sol_{a_1'}] J_{ST}(a_1', a_2^{m\prime} \cup a_2^f)$$

$$[b_1' := sol_{b_1'}] Prd_{b_1, b_1'}(U_{b_1}) \wedge [b_1' := sol_{b_1'}] J_{UV}(b_1', b_2^{m\prime} \cup b_2^f)$$

Partitioning the variables a_1 (resp. b_1) into a_1^m and a_1^f (resp. b_1^m and b_1^f) allows us to rewrite these last hypotheses as:

$$[a_1^{m\prime} := sol_{a_1'}] Prd_{a_1^m, a_1^{m\prime}}(S_{a_1}) \wedge [a_1' := sol_{a_1'}] J_{ST}(a_1', a_2^{m\prime} \cup a_2^f) \quad (\mathrm{H}_{21})$$

$$[b_1^{m\prime} := sol_{b_1'}] Prd_{b_1^m, b_1^{m\prime}}(U_{b_1}) \wedge [b_1' := sol_{b_1'}] J_{UV}(b_1', b_2^{m\prime} \cup b_2^f) \quad (\mathrm{H}_{22})$$

Reusing the proofs of $S_{a_1} \sqsubseteq_{J(a_1, a_2)} T_{a_2}$ and $U_{b_1} \sqsubseteq_{J(b_1, b_2)} V_{b_2}$ consists in checking that the solutions $sol_{a_1'}$ and $sol_{b_1'}$ are also a solution for the goal to prove. This means that the tuple $(sol_{a_1'}, sol_{b_1'})$ has to satisfy $(\mathrm{G}_{9.1})$. By definition of the parallel substitution, we know that: $a_1^m \cap b_1^m = \emptyset$. So, the substitution of the values $sol_{a_1'}$ and $sol_{b_1'}$ in $(\mathrm{G}_{9.1})$ gives:

$$[a_1' := sol_{a_1'} \| b_1^{f\prime} := sol_{b_1^{f\prime}}] Prd_{a_1^m, a_1^{m\prime}}(S_{a_1}) \quad (\mathrm{G}_{9.2})$$

$$[a_1^{f\prime} := sol_{a_1^{f\prime}} \| b_1' := sol_{b_1'}] Prd_{b_1^m, b_1^{m\prime}}(U_{b_1}) \quad (\mathrm{G}_{9.3})$$

$$[a_1' := sol_{a_1'} \| b_1' := sol_{b_1'}](d_1' = d_1) \quad (\mathrm{G}_{9.4})$$

$$[a_1' := sol_{a_1'} \| b_1' := sol_{b_1'}] J(a_1^{m\prime} \cup b_1^{m\prime} \cup d_1', a_2^{m\prime} \cup b_2^{m\prime} \cup d_2) \quad (\mathrm{G}_{9.5})$$

[7] $[x := y]P$ denotes the substitution of each free variable x of P by y.

One condition to prove ($G_{9.2}$) (resp. ($G_{9.3}$)) by reuse of (H_{21}) (resp. (H_{22})) is that the modified variables a_1^m and b_1^f (resp. a_1^f and b_1^m) must be disjoint. i.e:

$$\boxed{a_1^f \cap b_1^m = \emptyset \qquad and \qquad b_1^f \cap a_1^m = \emptyset}$$

The goal ($G_{9.4}$) is obvious because the variables d_1 denote the common unchanged variables of a_1 and b_1. Finally, we must be able to deduce the goal ($G_{9.5}$) from the current hypothesis environment, that is, to prove that:

$$\boxed{\begin{array}{r} (Trm(S_{a_1}) \wedge Trm(U_{b_1}) \wedge J(c_1, c_2) \wedge Prd_{a_2^m, a_2^{m\prime}}(T_{a_2}) \wedge \\ Prd_{b_2^m, b_2^{m\prime}}(V_{b_2}) \wedge [a_1' := sol_{a_1'}]Prd_{a_1^m, a_1^{m\prime}}(S_{a_1}) \wedge \\ [b_1' := sol_{a_1'}]Prd_{b_1^m, b_1^{m\prime}}(U_{b_1}) \wedge [a_1' := sol_{a_1'}]J_{ST}(a_1', a_2^{m\prime} \cup a_2^f) \wedge \\ [b_1' := sol_{b_1'}]J_{UV}(b_1', b_2^{m\prime} \cup b_2^f)) \rightarrow \\ [a_1' := sol_{a_1'} \| b_1' := sol_{b_1'}]J(a_1^{m\prime} \cup b_1^{m\prime} \cup d_1, a_2^{m\prime} \cup b_2^{m\prime} \cup d_2) \end{array}}$$

Conclusion: let $S_{a_1} \sqsubseteq_{J_{ST}(a_1, a_2)} T_{a_2}$ and $U_{b_1} \sqsubseteq_{J_{UV}(b_1, b_2)} V_{b_2}$ be two already proved refinements. Let $J(a_1 \cup b_1, a_2 \cup b_2)$ a predicate such that:

i. $(Trm(S_{a_1}) \wedge Trm(U_{b_1}) \wedge J(a_1 \cup b_1, a_2 \cup b_2)) \rightarrow J_{ST}(a_1, a_2)$
ii. $(Trm(S_{a_1}) \wedge Trm(U_{b_1}) \wedge J(a_1 \cup b_1, a_2 \cup b_2)) \rightarrow J_{UV}(b_1, b_2)$
iii. $a_1^f \cap b_1^m = \emptyset$
iv. $b_1^f \cap a_1^m = \emptyset$
v. $a_2^f \cap b_2^m = \emptyset$
vi. $b_2^f \cap a_2^m = \emptyset$
vii. Let a_1' and b_1' be solutions of the refinements $(S_{a_1} \sqsubseteq_{J_{ST}(a_1, a_2)} T_{a_2})$ and $(U_{b_1} \sqsubseteq_{J_{UV}(b_1, b_2)} V_{b_2})$ for given values of a_2' and b_2' respectively. If the values a_1' and b_1' satisfy :

$$\begin{array}{c} (Trm(S_{a_1}) \wedge Trm(U_{b_1}) \wedge J(a_1 \cup b_1, a_2 \cup b_2) \wedge Prd(T_{a_2}) \wedge Prd(V_{b_2}) \\ Prd(S_{a_1}) \wedge Prd(U_{b_1}) \wedge J_{ST}(a_1', a_2') \wedge J_{UV}(b_1', b_2')) \\ \rightarrow J(a_1' \cup b_1', a_2' \cup b_2') \end{array}$$

then the tuple (a_1', b_1') can be reused to prove: $(S_{a_1} \| U_{b_1}) \sqsubseteq_{J(a_1 \cup b_1, a_2 \cup b_2)} (T_{a_2} \| V_{b_2})$ for the same values of a_2' and b_2'.

4.4 Analysis of the results

1. Conditions (i) and (ii) are the first two necessary conditions of reuse. Indeed, if we want to adopt a reuse strategy in our proof process, then we must exhibit the gluing invariants of the refinements we would like to reuse in the refinement we are proving.
2. Conditions (iii, iv, v, vi) are related to the refinement of the parallel substitution. Indeed, if a proof is established in the case where a variable x is unchanged then it is obvious that this proof can only be reused in the same conditions (x must remain unchanged). This means that the set of variables modified by one substitution and the set of variables not changed by the other one must be disjoint.

3. Condition (*vii*) is the third necessary condition of reuse. It states that only the solutions a_1' and b_1', of $(S_{a_1} \sqsubseteq T_{a_2})$ and $(U_{b_1} \sqsubseteq V_{b_2})$ respectively, satisfying the global invariant can be reused.
4. If the two substitutions S_{a_1} and U_{b_1} are deterministic, then the condition (*vii*) always holds. Indeed, there exists only one value of $({a_1}', {b_1}')$ that satisfies the two basic refinements. This value is defined by $(Prd(S_{a_1}), Prd(U_{b_1}))$. Each possible solution of the refinement $(S_{a_1} \| U_{b_1}) \sqsubseteq_{J(a_1 \cup b_1, a_2 \cup b_2)} (T_{a_2} \| V_{b_2})$ must satisfy the two predicates $Prd(S_{a_1})$ and $Prd(U_{b_1})$ which have only one solution. So, the unique solutions of the basic refinements are also solutions for the parallel refinement.
5. If the predicates $J(a_1, a_2)$ and $J(b_1, b_2)$ are functional[8] on a_1 and b_1 respectively, then the condition (*vii*) holds. Indeed, each possible solution of the refinement $(S_{a_1} \| U_{b_1}) \sqsubseteq_{J(a_1 \cup b_1, a_2 \cup b_2)} (T_{a_2} \| V_{b_2})$ must satisfy $J(a_1 \cup b_1, a_2 \cup b_2)$. According to (*v*) and (*vi*) this solution satisfies $J(a_1, a_2)$ and $J(b_1, b_2)$. But, these two predicates have one unique solution. So, the unique solutions of the basic refinements are also solutions for the parallel refinement.
6. One may argue that it would be easier to directly establish the refinement proof than to check if some reuse conditions are satisfied. We don't think so. Indeed checking if a given value satisfies a formula is always easier than exhibiting the values themselves. Moreover, this verification can be automatically achieved. This implies that the refinement proof can be automatically discharged. It is an important benefit since provers generally fail to automatically prove existential formulae.

Whatever the B constructor, the reuse conditions are similar. We have also considered the case of operations whose refinement requires more than two basic refinement rules [17]. In conclusion, we have demonstrated that to reuse the proof of the refinement $S_{a_1} \sqsubseteq_{J(a_1, b_1)} T_{b_1}$, three conditions are required:

a. the gluing invariant of the refinement $S_{a_1} \sqsubseteq_{J(a_1, b_1)} T_{b_1}$ must be deduced from the proof environment.
b. the value, satisfying the refinement $S_{a_1} \sqsubseteq_{J(a_1, b_1)} T_{b_1}$, must satisfy the gluing invariant of the refinement to be proved.
c. for the parallel constructor, the set of variables modified by one substitution and the set of variables not changed by the other one must be disjoint.

An interesting consequence of this result is that it is possible to develop an automatic prover for a domain of applications where the reuse conditions are always satisfied. This is the subject of the next section.

5 An automatic prover dedicated to the refinement of data-intensive applications

In this section, we describe our approach for the development of an automatic prover dedicated to the refinement of data-intensive applications. Recall that

[8] A predicate $P(a, b)$, depending on two set of variables a and b, is functional on a iff for each value of b, there is a unique value of a that satisfies the predicate $P(a, b)$

our goal is to provide software engineers with a formal approach, based on the B refinement process, for the development of reliable data-intensive applications. Such an approach may be difficult to use if the users have to achieve by hand all the proofs necessary to establish the correctness of the code generated at the last step. Moreover, it is recognized that the proof phase requires significant skills. For this, we have consider the construction of a dedicated prover that makes this proof phase a push-button activity.

In the first subsection, we show that, within the database domain we are considering, all the gluing invariants of our refinement rules have the required characteristics to satisfy the conditions (a) and (b). Then, the construction of an automatic prover comprises two phases: the automation of the proof of the basic substitutions refinement and the automation of the proof of the B constructors refinement.

5.1 Satisfaction of the reuse conditions

As we can remark, the three conditions stated in the previous section depend closely on the substitutions and the gluing invariant features. As our refinement system is closed, only the rules defined in the base can be used. Let us examine why the conditions (a) and (b) are satisfied by the set of refinement rules we have defined.

The condition (a) is satisfied by construction of the global gluing invariant. Indeed this invariant is equal to the conjunction of the gluing invariants of the different elementary refinement rules which are applied.

The condition (b) is also satisfied. Indeed, each rule of our refinement process is characterised by a deterministic gluing invariant. In section 4, we have pointed out that this condition is sufficient to satisfy the second reuse condition.

Of course, the condition (c) depends on the way the user has specified the abstract substitution. For instance, the condition is not satisfied if he/she specifies a transaction that simultaneously modifies an attribute and uses its value in the update of another attribute. In this case, the proof must be achieved without reuse. Nevertheless, this is not a frequent case.

These results have a double interest. Indeed, the fact that the reuse conditions are fulfilled ensures that the solutions exhibited for the proof of the correctness of elementary refinements can be reused to prove the correctness of refinements built on the elementary refinements. This means that one are discharged from the search of the values of abstract variables to achieve the proof. In practice, this task may be rather difficult. Moreover, as the global invariant is the conjunct of the elementary invariants, establishing the global proof comes down to establishing the elementary proofs. This means that the proof trees of the elementary proofs can be reused as is in the proof tree of the global proof.

5.2 Refinement proof of basic substitutions

We have defined for each refinement rule a proof tactic that enables the automation of its correctness. The tactics are implemented using the prover of Atelier B. Recall that the proof of correctness of a refinement consists in exhibiting a value of a', associated to a given value b' satisfying $Prd(T_b)$, that satisfies the two predicates $Prd(S_a)$ and $J(a', b')$ (see page 4). However, all the abstract and concrete substitutions of the basic operations we consider are deterministic assignments of the form $(a := E)$. So, the searched value a' is given directly by the term $Prd(S_a)$ $(a' = E)$. In the same way, the value of b' is given by the term $Prd(T_b)$. Then, to prove the correctness of the refinement, we just need to check that the term $J(a', b')$ is true for these values of a' and b'. For example, to prove the goal of *Rule_Add*, page 7, we have to check that:

$$(f_1 \cup \{c \mapsto v(d)\}) = (v \circ (f \cup \{c \mapsto d\}))$$

Using Atelier B, we have defined a tactic that achieves the proof of this last formula. A tactic is an application of an ordered set of deduction rules. To prove the *Rule_Add rule*, we defined the following B tactic[9]:

$$tac\_Add \mathrel{\widehat{=}} \circ\_dist\_\cup;\ equal\_union;\ (comp\_ima;\ axio)^{+}$$

where:

- $\circ\_dist\_\cup$ states the distributivity property of the composition operator on the union one:

$$\frac{}{a \circ (b \cup c) = (a \circ b) \cup (a \circ c)}$$

- *equal_union* is a simplification rule. It gives a sufficient condition to demonstrate that $a \cup c = b \cup d$

$$\frac{a = b \qquad c = d}{a \cup c = b \cup d}$$

- *comp_ima* expresses the property of the composition operator on a function:

$$\frac{binhyp(f \in c \rightarrow d) \qquad e \in c}{(f \circ \{g \mapsto e\}) = \{g \mapsto f(e)\}}$$

  $binhyp(H)$ specifies a guard (condition) for the application of the considered rule. It enables the identification of the symbols used in the hypotheses and that don't appear in the goal (the symbols c and d in our case).
- *axio* enables to discharge the goals that are in the hypotheses:

$$\frac{binhyp(H)}{H}$$

Using the B tactic *tac_Add*, the proof tree associated to the correctness of *Rule_Add* is constructed as follows:

---

[9] $(r_1;\ r_2)$ means that the rule r_1 is applied first, then the rule r_2 is applied. r^{+} means that the rule (or an ordered set of rules) is applied as many times as possible.

$$\dfrac{\dfrac{\dfrac{}{f_1=(v\circ f)}\,axio \qquad \dfrac{\dfrac{}{d\in C}\,axio}{\{c\mapsto v(d)\}=(v\circ\{c\mapsto d\})}\,comp\_ima, v\in C\to T}{(f_1\cup\{c\mapsto v(d)\})=(v\circ f)\cup(v\circ\{c\mapsto d\})}\,equal\_union}{(f_1\cup\{c\mapsto v(d)\})=(v\circ(f\cup\{c\mapsto d\}))}\,\circ\_dist\_\cup$$

In the same way, we have defined another B tactic for the proof of the *Rule_Del* refinement rule.

5.3 Refinement proof of B constructors

As we have already noted, the correctness proof of a transaction refinement comprises the correctness proof of the basic operations that compose it and the refinement proof of the B substitutions constructors that relate these basic operations. In the previous subsection, we have shown, through the running example, how the correctness of the first category of proofs is automated by defining B tactics. In this subsection, we illustrate the automation of the second category.

At the abstract level, a database transaction is constructed on basic substitutions using a combination of **IF** constructors, parallel constructors (||) and non-deterministic constructors (**ANY**). In the following, we discuss the refinement and the correctness of the **IF** constructor, the reasoning is similar for the other constructors.

Using our refinement process, the conditional substitution (**IF** P **THEN** S **END**) is refined by rewriting the predicate P with respect to concrete variables, and by refining the substitution S by a substitution T. So, we obtain a concrete substitution of the form (**IF** Q **THEN** T **END**). According to the boolean values of P and Q, four proofs are raised by the proof obligations generator (GOP) of Atelier B:

- the first two proofs correspond to the cases where P and Q have opposite boolean values. These proofs are achieved by exhibiting that Q is a rewriting of P. In this case, the proof becomes trivially true because we have two contradictory hypotheses. For example, to achieve the proofs (2) and (3) of page 9, we have defined the following tactic:

  $$tac\_abs \mathrel{\widehat{=}} (replace;\ axio\_cont;\ axio^{+})$$

  where the rules *replace* and *axio_contradictory* are defined by[10]:

  $$\frac{b=(a\circ c)}{c=(a^{-1}\circ b)}\,replace \qquad\qquad \frac{band(binhyp(H),binhyp(\neg H)}{G}\,axio\_contradictory$$

- the two other proofs concern the cases where the predicates P and Q have the same boolean value. In these two cases, we have to prove the correctness of the refinement of the abstract substitution by the corresponding concrete

[10] $band(A, B)$ means that the prover searches for each possible hypothesis that matches A the corresponding hypothesis that matches B. The search stops when two hypotheses that match A and B respectively are found.

one. It is here that the reuse of the basic proofs previously elaborated take effect. Theses cases corresponds to the proofs (1) and (4) of the example of page 9.

>From a practical point of view, the specialized prover have been implemented within Atelier B as follows. We have created a *PatchProver* file in which different tactics are defined. When a *PatchProver* file is executed, the tactics are applied, one after the other, on each unproven proof obligation. As this process may be very time consuming, we relate each tactic to a particular kind of proof goal.

6 Conclusion and future works

In this paper, we have presented the approach that enables the development of an automatic prover dedicated to the refinement of database applications. The approach is based on a proof reuse strategy. In practice, it is frequent that a prover fails to achieve a proof without user interventions. However, if a proof reuse strategy is applied, what has been learned from previously computed proofs may guide the solver to automatically prove a larger amount of proofs. Another important benefit of reuse is resources (memory for example) and time saving. In the B refinement, for example, the gluing invariant may be very complex, and retrieving the value that satisfies it is not an obvious task. It is especially crucial in case of provers operating with limited resources or time as it is the case of the prover of the Atelier B [2] witch breaks down automatically after a given time.

We consider proof reuse within the context of refinement reuse. Firstly, necessary and sufficient conditions of proof reuse have been defined for the refinement of each B constructor. We store elementary refinements as quadruplets of abstract/concrete substitutions, gluing invariant and the refinement solution. Then, the resolution of a new elaborated refinement consists in retrieving, in our refinement base, the quadruplets whose abstract and concrete substitutions appear in the new refinement. It remains to select which basic solutions satisfy the current gluing invariant, and to check if the proof reuse conditions hold, which is not difficult to achieve.

As all reuse strategies, the usefulness of such proposition depends on the properties of the underlying solver but also on the domain which is considered. In general, a reuse strategy is successful if it is applied to a domain where its applicability conditions are often verified. In this paper, we have pointed out that, in data-intensive applications, these conditions most often hold. This result has allowed us to construct an automatic prover within the Atelier B prover.

The approach is developed in the framework of the B method and Atelier B. However it may be adapted to either another formal method or a different application domain, provided the following elements are defined: a systematic development strategy that provides a set of basic proofs for the considered application domain and a set of conditions to satisfy under which proof reuse is possible.

From a theoretical perspective, we have now to mechanically check that the demonstrations we have carried out to exhibit the reuse constraints are correct. This work requires a significant reflection about the prover to be used.

References

1. Abrial, J.R.: The B-Book, Cambridge University Press, 1996.
2. Clearsy. Atelier B, manuel de référence. available at http://www.atelierb.societe.com.
3. B-Core. B-Toolkit, on-line manual. Oxford, UK, available at http://www.b-core.com.
4. Barras, B., et al., The Coq Proof Assistant, Reference Manual (7.1), INRIA Rocquencourt, 2001.
5. Barthe, G., Pons, O.: Type Isomorphisms and Proof Reuse in Dependent Type Theory, Proceedings of FOSSACS'01, volume 2030, pages 57-71, Springer, 2001.
6. M. Blaha and W. Premerlani. *Object-Oriented Modeling and Design for Database Applications.* Prentice Hall, 1998.
7. Burdy, L., Meynadier, J-M.: Automatic Refinement, BUG Meeting, FM'99, Toulouse, France, September 1999.
8. S. Ceri. *Methodologies and Tools for Database Design.* Elsevier Science, 1983.
9. R. Laleau. On the interest of combining UML with the B formal method for the specification of database applications. In *ICEIS2000: 2nd International Conference on Enterprise Information Systems, Stafford, UK*, July 2000.
10. Laleau, R., Mammar, A.: A Generic Process to Refine a B Specification into a Relational Database Implementation, Int. Conf. ZB2000, Springer-Verlag, LNCS 1878, York, 2000. Extended version in the CEDRIC research report N 86.
11. R. Laleau and A. Mammar. An overview of a method and its support tool for generating B specifications from UML notations. In *ASE: 15th IEEE Conference on Automated Software Engineering, Grenoble, France.* IEEE Computer Society Press, September 2000.
12. R. Laleau and F. Polack. Specification of integrity-preserving operations in information systems by using a formal UML- based language. *Information and Software Technology*, 43:693–704, 2001.
13. R. Laleau: Conception et développement formels d'applications bases de données. Habilitation Thesis, CEDRIC Laboratory, Évry, France, 2002. Available at `http://cedric.cnam.fr/PUBLIS/RC424.ps.gz`
14. Luo, Z.: Coercive Subtyping in Type Theory, In CSL book,276-296, 1996. Also available from "citeseer.nj.nec.com/luo96coercive.html".
15. Magaud, N., Bertot, Y.: Changement de Représentation de Données dans le Calcul des Constructions Inductives. Research report, RR-4039, INRIA, France, October 2000.
16. Magaud, N., Bertot, Y.: Changement de Représentation de Structures de Données dans Coq: le cas des entiers naturels, In Proceedings of JFLA'2001.
17. Mammar, A.: Un environnement formel pour le développement d'applications bases de données. PhD thesis, CEDRIC Laboratory, CNAM, Evry, France, November 2002. Available at `http://cedric.cnam.fr/PUBLIS/RC392.ps.gz`
18. Pons, O.: Generalization in Type Theory Based Proof Assistants, In Proceedings of TYPES'00. Durham, United Kingdom December 2000.
19. Walther, C., Kolbe, T.: Proving Theorem by Reuse, Artificial Intelligence, 116:17-66, 2000.