Refining interfaces: the case of the B method

Article · July 2009

Source: arXiv

3 authors, including:

David Déharbe
ClearSy System Engineering
114 PUBLICATIONS **940** CITATIONS
SEE PROFILE

Anamaria Martins Moreira
Federal University of Rio de Janeiro
62 PUBLICATIONS **222** CITATIONS
SEE PROFILE

Refining interfaces: the case of the B method

David Déharbe, Bruno E.G. Gomes, and Anamaria Martins Moreira

Universidade Federal do Rio Grande do Norte
Departamento de Informática e Matemática Aplicada
Natal, RN, Brazil {`david,bruno,anamaria`}`@dimap.ufrn.br`

Abstract. Model-driven design of software for safety-critical applications often relies on mathematically grounded techniques such as the B method. Such techniques consist in the successive applications of refinements to derive a concrete implementation from an abstract specification. Refinement theory defines verification conditions to guarantee that such operations preserve the intended behaviour of the abstract specifications. One of these conditions requires however that concrete operations have exactly the same signatures as their abstract counterpart, which is not always a practical requirement. This paper shows how changes of signatures can be achieved while still staying within the bounds of refinement theory. This makes it possible to take advantage of the mathematical guarantees and tool support provided for the current refinement-based techniques, such as the B method.

1 Introduction

Java Card [1] is a state-of-the-art technology that provides a programming environment for smart cards that is compatible with the Java programming language and its underlying platform. Due to the limited processing power of the chips found on smart cards, Java Card components are small and require few resources. They thus provide an interesting testbed for formal approaches to software design such as the B method [2]. The B method implements a rigorous model-driven design approach to derive software from a functional specification through a series of stepwise refinements. It has mature tool support and has been successfully applied by, e.g. the railway industry, to develop the software of safety-critical systems.

The goal of the *Bsmart* project [3] is to develop a customized version of the B method for the development of Java Card software components, as well as the corresponding tool support (as an Eclipse plug-in [4]). Applications using smart cards have a client-server approach, where the server is a Java Card component that provides access to the smart card services and the client is (usually) developed in Java and accesses such services through a mechanism such as a remote method invocation. Although based on the same programming paradigms, the type system of Java Card is much simpler and restricted than that of Java. Java client software often requires services in a richer type system than that provided by the Java Card services, and the APIs need to be adapted. So, in order to be able to include a richer type system in Bsmart, it appears necessary to include a refinement step corresponding to such interface adaptation.

Unfortunately, the concept of refinement used in the B method does not allow for modification of the signature of the operations that compose such interfaces [2]. Retrenchment [5] is a much more flexible concept of model transformation, that includes changes in signature operations and, consequently, in component interfaces. The scope of retrenchment is however much larger than simple interface changes, and also includes handling much deeper model transformations, such as, e.g. strengthening pre-conditions of operations. This extra flexibility allows implementations that exhibit behaviours that are not in the original functional specification, which may not be desirable in a rigorous model-based development. Also, although the proponents of retrenchment have developed syntactic extensions to the B method to include such transformation, these extensions do not yet benefit from the same level of tool support as refinement.

The goal of this paper is to show a solution to interface changes that fits within the classical theory of refinement. Thus, it does not require employing retrenchment and introducing model transformations that result in executions that are not modeled in the initial functional specification. In addition, the solution proposed in this paper consists in model transformations that are fully compatible with existing tool support for the B method. Indeed, we have defined the generic refinement pattern, as well as an instance thereof, in B itself and have used existing tools to prove their correctness.

Several authors have related interface changes with refinement [6–9], however none of thes works is related to the B method; also they change the verification conditions associated to refinement. In [10], an approach similar to ours is presented in the context of component-based development; however they do not go so far as to present a refinement pattern as detailed as the one presented in this paper.

Plan of the paper. Section 2 briefly introduces the B method and introduces an example that will be used throughout the paper to illustrate the different model transformations. Also, the main concepts of retrenchments are exposed and discussed in Section 3. Section 4 then presents the refinement pattern to introduce interface changes and a model transformation instantiating this pattern is presented in Section 5. Finally, conclusions and future work are presented in Section 6.

2 Model-driven development with B

The B method for software development [2, 11] is a model-driven development method based on formal models and formally verified derivations or refinements. It provides the B *Abstract Machine Notation* (AMN) to represent models at different levels of abstraction, based on first order logic, integer arithmetic and set theory. These different levels of abstraction of a model must be related by formally proved refinements.

Industrial tools for the development of B based projects have been available for a while now [12, 13], with specification and verification support as well as some project management tasks and support for team work. More recently, various academic and/or open source tools have spread, and Atelier B [12] has become free of charge, increasing the popularity of the method and the variety of its uses.

2.1 The B development process

A B specification is structured in components. The initial model from which the software development process initiates may be modularly composed of one or more *MACHINE*s. Such models must be proved satisfiable (i.e. that they have an implementation) and consistent with respect to some specified properties (namely, the INVARIANT of each MACHINE).

Once an abstract model is proved consistent, it may be used as input for a series of (optional) refinements. The result of each refinement step for each MACHINE is a new (usually less abstract) module classified as a *REFINEMENT*. The obtained refined model is then proved correct with respect to the abstract model. This is done modularly, by proving the correctness of each REFINEMENT component with respect to its corresponding machine and to all intermediate REFINEMENT components in between the abstract MACHINE and the REFINEMENT being verified.

Eventually, a final refinement takes place, which gives origin to a B *IMPLEMENTATION*, a special kind of refinement from which code in a programming language can be generated. The verification of the model at the IMPLEMENTATION level is carried out similarly as for refinements, with the addition of the so-called *B0 check*, which is responsible for verifying that the constructs in each IMPLEMENTATION module are compatible with the used code generator.

Finally, B IMPLEMENTATIONs are used as input for code generation in some programming language (e.g., C, Ada or Java). If all verifications were discharged, and assuming the correctness of the code generator, this generated code satisfies the stated properties of the abstract model.

2.2 Components of a model in the B notation

A B component contains two main parts: a state space definition and a set of transitions. The state space is specified as a logic formula called the invariant. Transitions are specified by means of *operations*; generally, each operation may take arguments and return results corresponding to a desired functionality of the system. The set of initial states is specified as a special operation (without parameters nor results) called the *initialisation*. A B component may additionally contain clauses in many forms (parameters, constants, assertions). Such clauses are not essential in the B language, but are useful to make specifications and proofs shorter or more readable.

The specification of the state components appears in the VARIABLES and INVARIANT clauses. The former enumerates the state components, and the latter defines restrictions on the possible values they can take.

For the specification of a module's operations, B offers a language of so-called *generalized substitutions*, "imperative-like" constructions with translation rules that define their semantics as the effect they have on the values of any expression on the (global or local) variables to which they are applied. The semantics of the substitutions is defined by the *substitution calculus*, a set of rules stating how the application of the different forms substitution rewrite to formulas in first-order logic. Let S denote a substitution, E an expression, then $[S]E$ denotes the result of applying S to E.

Operations are composed of a pre-condition P and a substitution S. Syntactically, this is expressed as **PRE** P **THEN** S **END**. In this construct, P specifies the bounds of application of the operation, and S specifies what transformations will be applied to the state, as well as how the operation results (if any) are computed. Operations also have optional parameters and results. The pre-condition P must establish at least typing constraints on the parameters and the substitution S define the value of the results. To establish that an operation does not drive the component from a valid state to an invalid state, one must show that the operation, whenever applied in a state that satisfies the pre-condition, maintains the invariant, i.e.

$$I \land P \Rightarrow [S]I.$$

The simplest substitution in the B language is $v := E$ where v is a variable and E denotes some expression. The semantics is defined as:

$$[v := E]P \Leftrightarrow P\langle v \leftarrow E\rangle,$$

i.e. all free occurrences of v in P are replaced by E.

Another substitution that is used in the rest of the paper is a form of non-deterministic assignment $v :\in V$, where v is allowed to take any value in the set V. The semantics is:

$$[v :\in V]P \Leftrightarrow \forall x \bullet (x \in V \Rightarrow P\langle v \leftarrow x\rangle),$$

where x is a fresh variable. Note that, for such substitution to be well-defined, one must show that V is not an empty set.

2.3 Example of a B Machine

In this section, we present a simple example of a B model that will be used throughout the paper. Our example is that of a simple counter (Figure 1). In the next sections, this abstract specification, which intends to specify the Application Programming Interface (API) of a counter service, will be refined with the intention of having this service offered by a Smart Card running Java Card. Some difficulties in this process motivate our proposal.

```
MACHINE    JCounter
SEES    JInt
VARIABLES    value
INVARIANT    value ∈ JINT
INITIALISATION    value := jint_of (0)
OPERATIONS
    increment (vv)=
    PRE    vv ∈ JINT ∧ sum_jint(value, vv) ∈ JINT
    THEN    value := sum_jint(value, vv)
    END;
    decrement (vv) =
    PRE    vv ∈ JINT ∧ subt_jint(value, vv) ∈ JINT ∧ (value - vv) ≥ 0
    THEN    value := subt_jint(value, vv)
    END;
    cc ← getCounterValue = cc := value
END
```

Fig. 1. The counter machine

In the *JCounter* machine, the variable *value* is the state component that stores the actual value of the counter. This variable is typed as *JINT*, an integer set defined in the *JInt* machine that corresponds to, e.g., the *int* type of Java language (this machine belongs to a library of B models, under development by our group, to support the formal development of Java and Java Card software). The machine *JInt*, not shown in the paper, also defines arithmetic operations on this set so that they operate within the range for a Java value of type *int*. The included *JInt* machine uses these properties in the definition of functions that can be used in substitution of the B operators in the body of an operation. The specification *JCounter* comprises three operations to increment, decrement and query the counter. In the last operation, the pre-condition is omitted, which is interpreted as a trivial pre-condition (i.e. *TRUE*). Note that, in the body of the operations, we use the arithmetic functions defined in *JInt* machine instead of the B operators for integers. This means that we could be talking of any other type of data not directly available in B.

2.4 Refinements in B

Refinements, which are central to the proposal of this paper, play a very important role on the B method. They are responsible for the creation of a hierarchy of models where each model is proved to be compliant, according to the B refinement rules [2], to the previous (more abstract) one in the chain. We briefly present these rules in the following.

1. exactly the same number of operations
2. exactly the same operation interfaces (names, parameters and results)
3. each concrete operation must satisfy the classical rules stating that:

(a) it must be applicable whenever its abstract counterpart is (the satisfaction of the precondition of the abstract operation must lead to the satisfaction of the precondition of the concrete operation).
(b) when the abstract operation is applicable and the concrete operation is applied instead, the observed behavior must be compatible with one of the behaviors specified in the abstract operation.

3 Retrenchment

The refinement rules presented in the previous section aim to guarantee that any implementation of the concrete model can be transparently used as an implementation of the abstract model, but they are sometimes considered an unnecessary burden to refinement based development [5]. Refinement rules can hinder its adoption on the design of a wide range of real world applications, as the differences between an elegant abstract model and a concrete model, where implementation needs begin to show up, may not fit the refinement framework. If we consider, for instance, our counter example of section 2.3 and the need to implement it in a platform where only short integers are available (this may happen in some smart cards), there will be a problem with the operations' interfaces, which are supposed to communicate regular length integers. Changing the abstract specification to make the development fit in the refinement framework is not a good approach, as it degrades reusability and requires verifying the abstract level again. Retrenchment and the approach presented in this paper do not need any changes in the abstract specification.

With the main motivation of extending the applicability of formal development techniques to a wider range of applications, Banach and Poppleton have proposed a technique called *retrenchment* [5], a formal approach to model-driven design that imposes less constraining rules than refinement.

Indeed, with retrenchment it is possible to have stronger preconditions and/or weaker post-conditions in an operation, to change an operation interface and to transfer behavior from state components to I/O or vice-versa. In [5] the authors present the theory and its applicability and demonstrate how to incorporate it as an extension of the B method. In the following we present a brief explanation of this extension, concentrating on the possibility to change an operation's interface during the development process, the feature which is the focus of this paper.

3.1 Retrenchment in B

In Banach and Poppleton proposal, a retrenchment is a B machine with the addition of: (1) a RETRIEVES clause, to specify the retrieval relation, relating abstract and concrete variables[1], and (2) *ramified generalized substitutions*, con-

---

[1] Unlike B refinements, where the local invariant and the relation between the abstract and concrete states (retrieve relation) are both specified in the INVARIANT predicate, in a retrenchment module, the INVARIANT only specifies the more concrete state variables. The relation between the retrenching and the retrenched states is placed in the RETRIEVES clause.

structed with the clauses LVAR, WITHIN and CONCEDES, which extend each operation's generalized substitution, specifying the situations where the concrete operation fails to refine the abstract one (Figure 2).

All the elements to describe a B refinement are available to define a retrenchment: for instance, set definitions clause (SETS) and all the clauses for machine composition (SEES, INCLUDES, USES, PROMOTES and EXTENDS) can be used as in a traditional B module.

MACHINE $M_A(pm_M)$
CONSTRAINTS $P_A(pm_A)$
...
VARIABLES v_A
INVARIANT
  $Inv(v_A)$
INITIALISATION
  $Init(v_A)$
OPERATIONS
  $r_A \longleftarrow OP_A(p_A) =$
    $S_A(v_A, p_A, r_A)$
END

MACHINE $M_R(pm_R)$
CONSTRAINTS $P_R(pm_R)$
...
VARIABLES v_R
INVARIANT
  $Inv(v_R)$
RETRIEVES
  $Ret(v_A, v_R)$
INITIALISATION
  $Init(v_R)$
OPERATIONS
  $r_R \longleftarrow OP_R(p_R) =$
  BEGIN
    $S_R(v_R, p_R, r_R)$
  LVAR
    R
  WITHIN
    $W(p_A, p_R, v_A, v_R, R)$
  CONCEDES
    $C(v_A, v_R, r_A, r_R, R)$
END

Fig. 2. classical B machine (left) and retrenchment machine (right)

The LVAR clause is optional, and may be used to declare variables whose scope is the WITHIN and CONCEDES clauses. When present, these variables must be typed and restricted in the WITHIN clause, which may also strengthen the operation's precondition. The CONCEDES clause in turn possibly weakens the post-condition of the operation.

The role of these additional clauses can be more precisely described through the definition of retrenchment proof obligations, presented in the following section. Then, in Section 3.1, we use a small example to illustrate how retrenchment works in practice.

Retrenchment proof obligations Retrenchment proof obligations can be classified as local proof obligations, when only dealing with local data, and joint proof obligations, when addressing both the retrenched and retrenching components.

The local proof obligations of a retrenchment module are the same as those for a regular B machine: establishment of the invariant by the initialisation; preservation of the invariant by the operations, when applied to states where their preconditions are satisfied. By discharging these obligations, one guarantees the internal consistency of the module.

The joint proof obligations concern initialization and operations. The initialisation joint proof obligation is similar to that of a refinement, except for the fact that it is the satisfaction of the RETRIEVES predicate, instead of the INVARIANT, that is checked:

$$P_A(pm_A) \wedge P_R(pm_R) \Rightarrow [Init(v_R)]\neg[Init(v_A)]\neg Ret(v_A, v_R)$$

The proof obligations for the operations are the most relevant to the retrenchment framework. For each operation, correctness verification is conditioned to situations where: regular B constraints ($P_A(pm_A)$ and $P_R(pm_R)$), abstract and concrete invariants ($Inv(v_A)$ and $Inv(v_R)$) and the retrieve relation ($Ret(v_A, v_R)$) are satisfied; the concrete operation terminates ($trm(S_R(v_R, p_R, r_R))$) and the conditions stated on the WITHIN clause $W(p_A, p_R, v_A, v_R, A)$ are also satisfied. The first conditions are similar to those in a refinement proof obligation. It is important to notice that, differently from refinement, which requires correctness in each situation where the abstract operation terminates, it is the termination of the concrete operation that conditions the verification.

On the other hand, on the right hand side, we have the option of not satisfying the retrieve relation (i.e., having a concrete behaviour which does not correspond to a specified abstract behaviour) as long as the predicate in the CONCEDES clause is satisfied.

$$\begin{aligned}
&P_A(pm_A) \wedge P_R(pm_R) \wedge (Inv(v_A) \wedge Ret(v_A, v_R) \wedge Inv(v_R)) \wedge \\
&\quad trm(S_R(v_R, p_R, r_R)) \wedge W(p_A, p_R, v_A, v_R, R)) \Rightarrow trm(S_A(v_A, p_A, r_A)) \wedge \\
&\qquad [S_R(v_R, p_R, r_R)]\neg[S_A(v_A, p_A, r_A)]\neg(Ret(v_A, v_R) \vee C(v_A, v_R, r_A, r_R, R))
\end{aligned}$$

Retrenching JCounter In this section, we apply retrenchment in the formal development of the counter service of section 2.3 for a version of the Java Card platform without support for the Java type *int* (32-bit integers).

The architecture of a Java Card application is composed by host-side software and server-side software. The host side is developed in standard Java and requests the services supplied by the server application, called *applet*. The latter resides inside the smart card chip, which provides a computer with limited memory resources and processing power. Moreover, the Java Card language is much more limited than Java (for instance, it has a smaller set of basic types).

We assume that the smart card will provide the token counter service, that will be used by host-side applications written in Java. The development starts with the specification of the Java API that will be available to host-side clients (Figure 1). The obtained retrenchment, with different operation signatures than those of the abstract machine, is shown in Figure 3.

```
MACHINE JCounter_ret
RETRENCHES JCounter
SEES JInt, JCInt, InterfaceContext
VARIABLES cvalue
INVARIANT cvalue ∈ JCInt
RETRIEVES value = jint_of_jcint (value)
INITIALISATION cvalue := jcint_of (0)
OPERATIONS
    increment ( cvv ) =
    BEGIN
        PRE cvv ∈ JCINT ∧ sum_jcint(cvalue, cvv) ∈ JCINT
        THEN cvalue := sum_jcint(cvalue, cvv)
        END
    WITHIN vv = jint_of_jcint (cvv)
    END;
    decrement ( cvv ) =
    BEGIN
        PRE cvv ∈ JCINT ∧ subt_jcint(cvalue, cvv) ∈ JCINT ∧
            subt_jcint(cvalue, cvv) ≥ 0
        THEN cvalue := subt_jcint(cvalue, cvv)
        END
    WITHIN
        vv = jint_of_jcint (cvv)
    END;
    cc ← getCounterValue =
    BEGIN
         ccc = cvalue
    CONCEDES
        cc = jint_of_jcint (ccc)
    END
END
```

Fig. 3. A retrenchment of JCounter

As in our example, the basic data type *int* is not available, one possible solution is to represent it as a combination of available types, such as *short*. This representation is defined in the *JCInt* component (not shown) which defines the *JCINT* type, operators such as addition (*sum_jcint*) and subtraction (*subt_jcint*), and a type cast operation (*jcint_of*) to generate *JCINT* values from regular B integer values. The operations in *JCounter_ret* machine of Figure 3 run completely on this domain. This can be seen when observing the substitutions that specify the behaviour of each operation.

JCounter_ret also imports, through the **SEES** construct, *JINT* and *InterfaceContext* (described in Section 5), which, as one can see, only appear in the clauses related to retrenchment where they are used to specify the relation between specifications (*JCounter* and *JCounter_ret*). *jint_of_jcint* is a bijection, defined in *InterfaceContext* associating each Java integer to its Java Card representation. It is used in four different places: to specify the retrieve relation as it would regularly be done in a refinement; and in each operation, to associate each concrete parameter or result to its abstract counterpart. In a refinement, because there can be no changes in interfaces, this association is done automatically and does not need to be stated.

3.2 Some Notes on Retrenchment

Although retrenchment could be an attractive alternative to strict refinement for some developments, its adoption is currently not expressive and there is not yet a mature tool support for it. An academic initiative in this direction is the Frog tool [14], developed as part of the PhD thesis of Frasier [15]. The tool proposes a framework to mechanize the support for retrenchment. Initially the Z [16] notation was used as mathematical notation and the proof obligations were generated to the Isabelle theorem prover. But as the proposal of the framework is to be extensive, one can use it to configure its own formal model based development.

In the next section of the paper, we describe our solution to the problem of interface adaptation and type changing between models without going out the refinement theory using the B method.

4 Interface adaptation as refinement

This section describes a way how model transformations consisting of a modification in the signature of operations, can be performed by means of refinement. This transformation is presented as a refinement pattern [17] written and developed with the B method itself. Such pattern will then be instantiated in Section 5 for a simple software development for the Java Card platform.

4.1 A schematic specification in B

We first present the schema of a specification model in B. This schema is described in the B language itself as a component named API_A, that is presented in Figure 4. The types, sets and relations employed in the machine API_A are defined in the component $Context_A$, presented in Figure 5. Note that, for the sake of conciseness, the API_A machine only includes the clauses that provide the essence of what is a B model, namely a set of states, constrained by an invariant predicate, a set of transitions and initial states, both specified by means of substitutions. So, while there is no parameters, constants and sets in this pattern machine, the generality of the approach is thus not compromised.

A B component modelling a system has a state, and it is represented here as a single variable v_A, of type $type_A$ (defined in $Context_A$). The valid states are identified by the set inv_A and the initial states by the set $init_A$.

The transitions of the system are modelled by a single operation, named $operation_A$. The parameters of the operations are represented by p_A and its results by r_A. In the general case, an operation may have a precondition that depends on the state and parameters. It is here specified by means of the set pre_A. The possible next states and output values are chosen non-deterministically amongst the sets of values denoted stf_A and ouf_A respectively; both depend on the state variable and the operation parameter.

```
MACHINE  API_A
SEES  Context_A
VARIABLES  v_A
INVARIANT
  v_A ∈ type_A ∧ v_A ∈ inv_A
INITIALISATION
  v_A :∈ init_A
OPERATIONS
  r_A ⟵ operation_A(p_A) =
  PRE  p_A ∈ type_A ∧ (v_A, p_A) ∈ pre_A  THEN
    v_A :∈ stf(v_A, p_A) || r_A :∈ ouf_A(v_A, p_A)
  END
END
```

Fig. 4. A pattern for an abstract specification in B

In order to be able to prove the validity of the verification conditions of the component API_A, the objects defined in $Context_A$ need to satisfy a number of constraints, that are stated in its **PROPERTIES** clause.

The first five constraints are typing conditions, the next two constraints state that the domain of the state transition and output relations must contain the valid states and operation parameters. The last two constraints must be also satisfied to guarantee that all the reachable states of the component are also valid states (i.e. in the set representing the invariant).

Assume now that the component API_A is to be refined by a component API_C such that the data carried by state variables, operations parameters and results may be different. In the following, the objects in the component API_C will be here designated as the objects in the component API_A, with the A subscript substituted by the C subscript. For instance, the signature of the operation in the machine API_C is:

$$r_C \longleftarrow operation_C(p_C) =$$

where p_C satisfies $p_C \in type_C \wedge p_C \in pre_C$. Since the refinement of operations must preserve their signature, it is necessary to propose a workaround, such as

MACHINE $Context_A$
SETS $type_A$
CONSTANTS
stf_A, (* state transition function *)
ouf_A, (* output function *)
inv_A, (* state invariant *)
$init_A$, (* initial states *)
pre_A (* operation precondition: depends on state and parameter *)
PROPERTIES
$inv_A \subseteq type_A \quad \wedge$
$init_A \subseteq type_A \quad \wedge$
$pre_A \subseteq type_A \times type_A \quad \wedge$
$stf_A \subseteq (type_A \times type_A) \leftrightarrow type_A \quad \wedge$
$ouf_A \subseteq (type_A \times type_A) \leftrightarrow type_A \quad \wedge$
$inv_A \lhd pre_A \subseteq \mathbf{dom}(stf_A) \quad \wedge$
$inv_A \lhd pre_A \subseteq \mathbf{dom}(ouf_A) \quad \wedge$
$init_A \subseteq inv_A \quad \wedge$
$stf_A[inv_A \lhd pre_A] \subseteq inv_A$
END

Fig. 5. Component defining the objects used in component API_A

retrenchment does. In the next section we show a refinement pattern that makes it possible to use operations with a different signature in a refinement.

4.2 A refinement pattern for signature changes

The main idea that underlies the pattern is to use an interface adapter (see Figure 6). Note that this refinement is solely responsible for interfacing the two components and is not meant to introduce other design decisions such as reducing non-determinism, or precondition weakening.

The API_A component is refined by a component API_r. This refinement includes an instance of the component API_C, and the *gluing* invariant establishes the relationship between the state of API_A and the state of API_C. In API_r, the operation has the same signature as in API_A. It consists of a three-step sequence. First the value of the parameter p_A is translated to corresponding value of type $type_C$ and the result is stored in variable *to*. Second, $operation_C$ is applied to *to* and the result is stored in a variable *from*. The value of *from* is then converted back to $type_A$ and returned.

The conversion functions between $type_A$ and $type_C$ are declared and specified in the component $Context_I$, shown in Figure 7. The first two properties define the conversion functions *AofC* and *CofA* as total bijective functions. The third property constrains that they inverse each other. The properties numbered 4 to 7 constrain the translation functions to preserve the invariant states, the initial states, and the legal operation parameter values. The properties 8 to 11 further constrain that they preserve the state transition and output relations.

```
REFINEMENT $API_r$
REFINES $API_A$
SEES $Context_A, Context_C, Context_I$
INCLUDES $API_C$
INVARIANT $v_A = AofC(v_C)$
OPERATIONS
  $r_A \longleftarrow operation_A(p_A) =$
  VAR $to, from$ IN
    $to := CofA(p_A);$
    $from \longleftarrow operation_C(to);$
    $r_A := AofC(from)$
  END
END
```

Fig. 6. Schematic refinement that accommodates signature changes

Atelier B [12], an IDE for the B method, has been used to develop this pattern. To show the correctness of the development with the provers of Atelier-B, we introduced (and proved) the properties listed in "assertions" section.

5 Case study

In this section, we apply the refinement pattern described in Section 4 in the formal development of a Java Card implementation of the Counter specification presented in Section 2.3 and contrast it to the retrenchment approach exposed in Section 3.1.

As seen, a change in the interface of the operations is required, and in this section the refinement pattern of Section 4 is applied.

The *JCCounter* machine (Figure 8) provides the same services as the *JCounter* machine, but with its interface and typing restrictions compatible with the types of Java Card. In Java Card, the type *int* is not built-in and needs to be programmed, e.g. as a pair of short integers. This representation is defined and named by *JCINT* in a library machine called *JCInt* (not detailed in this paper). Note that the machine *JCCounter* is also the initial model of a B development to provide an implementation of the card-side component.

The functions mapping the values of the abstract (Java) and concrete (Java Card) types are defined in the *InterfaceContext* machine (see Figure 5). This machine also contains some corollaries in the assertions clause. These additional properties are useful to simplify interactive proofs of the development. These functions are essential to instantiate the refinement pattern to *JCounter*.

Finally, as a last step, the refinement itself, called *JCCounter_ref*, is also obtained by instantiation of the pattern and is presented in Figure 10. The development of this case study was also performed and verified with Atelier B [12].

MACHINE $\mathit{Context}_I$
SEES $\mathit{Context}_A, \mathit{Context}_C$
CONSTANTS $\mathit{AofC}, \mathit{CofA}$
PROPERTIES
$\mathit{AofC} \in \mathit{type}_C \rightarrowtail\mathrel{\mkern-14mu}\twoheadrightarrow \mathit{type}_A \quad \wedge$ 1
$\mathit{CofA} \in \mathit{type}_A \rightarrowtail\mathrel{\mkern-14mu}\twoheadrightarrow \mathit{type}_C \quad \wedge$ 2
$\mathit{CofA}^{-1} = \mathit{AofC} \quad \wedge$ 3
$\forall a \bullet (a \subseteq \mathit{type}_A \wedge a \subseteq \mathit{inv}_A \Rightarrow \mathit{CofA}[a] \subseteq \mathit{inv}_C) \quad \wedge$ 4
$\forall c \bullet (c \subseteq \mathit{type}_C \wedge c \subseteq \mathit{inv}_C \Rightarrow \mathit{AofC}[c] \subseteq \mathit{inv}_A) \quad \wedge$ 5
$\forall c \bullet (c \subseteq \mathit{type}_C \wedge c \subseteq \mathit{init}_C \Rightarrow \mathit{AofC}[c] \subseteq \mathit{init}_A) \quad \wedge$ 6
$\forall v_a, p_a \bullet (v_a \subseteq \mathit{type}_A \wedge p_a \subseteq \mathit{type}_A \wedge v_a \times p_a \subseteq \mathit{pre}_A \Rightarrow \mathit{CofA}[v_a \times p_a] \subseteq \mathit{pre}_C) \quad \wedge$ 7
$\forall v, p \bullet (v \in \mathit{type}_A \wedge p \in \mathit{type}_A \wedge (v, p) \in \mathbf{dom}(\mathit{stf}_A) \Rightarrow$ 8
$\quad (\mathit{CofA}(v), \mathit{CofA}(p)) \in \mathbf{dom}(\mathit{stf}_C)) \quad \wedge$
$\forall v, p \bullet (v \in \mathit{type}_A \wedge p \in \mathit{type}_A \wedge (v, p) \in \mathbf{dom}(\mathit{stf}_A) \Rightarrow$ 9
$\quad \mathit{CofA}[\mathit{stf}_A[\{(v, p)\}]] = \mathit{stf}_C[\{(\mathit{CofA}(v), \mathit{CofA}(p))\}]) \quad \wedge$
$\forall v, p \bullet (v \in \mathit{type}_A \wedge p \in \mathit{type}_A \wedge (v, p) \in \mathbf{dom}(\mathit{ouf}_A) \Rightarrow$ 10
$\quad (\mathit{CofA}(v), \mathit{CofA}(p)) \in \mathbf{dom}(\mathit{ouf}_C)) \quad \wedge$
$\forall v, p \bullet (v \in \mathit{type}_A \wedge p \in \mathit{type}_A \wedge (v, p) \in \mathbf{dom}(\mathit{ouf}_A) \Rightarrow$ 11
$\quad \mathit{CofA}[\mathit{ouf}_A[\{(v, p)\}]] = \mathit{ouf}_C[\{(\mathit{CofA}(v), \mathit{CofA}(p))\}])$
ASSERTIONS
$\mathit{AofC}^{-1} = \mathit{CofA} \quad \wedge$
$\mathbf{dom}(\mathit{AofC}) = \mathit{type}_C \quad \wedge$
$\mathbf{dom}(\mathit{CofA}) = \mathit{type}_A \quad \wedge$
$\forall a, c \bullet (a \in \mathit{type}_A \wedge c \in \mathit{type}_C \Rightarrow ((\mathit{AofC}(c) = a) \Leftrightarrow (c = \mathit{CofA}(a)))) \quad \wedge$
$\forall v_A, p_A \bullet (v_A \in \mathit{type}_A \wedge p_A \in \mathit{type}_A \wedge (v_A, p_A) \in \mathit{inv}_A \lhd \mathit{pre}_A \Rightarrow$
$\quad \mathit{CofA}[\mathit{stf}_A[\{(v_A, p_A)\}]] \subseteq \mathit{inv}_C) \quad \wedge$
$\forall s \bullet (s \subseteq \mathit{type}_A \Rightarrow \mathit{AofC}[\mathit{CofA}[s]] = s) \quad \wedge$
$\forall s \bullet (s \subseteq \mathit{type}_C \Rightarrow \mathit{CofA}[\mathit{AofC}[s]] = s)$
END

Fig. 7. Constraints to establish the refinement pattern for signature changes

```
MACHINE        JCCounter
SEES      JCInt
VARIABLES       jc_value
INVARIANT       jc_value ∈ JCINT
INITIALISATION       jc_value := jcint_of(0)
OPERATIONS
    jc_increment (vv)=
    PRE       vv ∈ JCINT ∧ sum_jcint(jc_value, vv) ∈ JCINT
    THEN       jc_value := sum_jcint(jc_value, vv)
    END; ...
    cc ← jc_getCounterValue =
        cc := jc_value
END
```

Fig. 8. The JCCounter machine

```
MACHINE        InterfaceContext
SEES      JInt,      JCInt
CONCRETE_CONSTANTS        jint_of_jcint, jcint_of_jint
PROPERTIES
    jint_of_jcint ∈
        JCINT ⤀ JINT ∧
        jint_of_jcint = λ (hi, lo).( (hi , lo ) ∈ JCINT | hi × 65536 + lo) ∧
    jcint_of_jint ∈
        JINT ⤀ JCINT ∧
        jcint_of_jint = λ (ii).(ii ∈ JINT | ((ii ÷ 65536 ), (ii   mod  65536)))

ASSERTIONS
      jint_of_jcint ⁻¹ = jcint_of_jint ∧
      dom (jint_of_jcint) = JCINT ∧
      dom (jcint_of_jint) = JINT

END
```

Fig. 9. The InterfaceContext machine

6 Conclusions

The B method provides a simple yet rigorous approach to model-driven design of software. Starting from an initial functional model of the requirements, additional requirements and implementation decisions are introduced as a sequence of refinements. For each refinement, proof obligations are generated; proving such verification conditions provides a formal guarantee that the initial specification is indeed an abstract of model of each successive refinement.

In the B method, the operations of a refinement must have the same signature as that of the refined module, and by transitivity, to that of the initial model. This limitation causes problems in software developments where com-

```
REFINEMENT    JCounter_ref
REFINES    JCounter
SEES    JInt, JCInt, InterfaceContext
INCLUDES    JCCounter
INVARIANT    value = jint_of_jcint(jc_value)
OPERATIONS
  increment ( vv ) =
  VAR    to
  IN    to := jcint_of_jint(vv);
    jc_increment(to )
  END;
   ...
  cc ← getCounterValue =
  VAR    from
  IN    from ← jc_getCounterValue;
        cc := jint_of_jcint(from)
  END
END
```

Fig. 10. The adapter refinement of counter machine

ponent interfaces must be adapted to accommodate, e.g. incompatibilities in programming languages.

Retrenchment provides a formal framework to perform model transformation that is much more flexible than refinement and, in particular, accommodates interface changes. However one may argue that the flexibility offered by retrenchment is too generous, to the point that it may produce implementations that do not conform to the initial functional specification. Indeed retrenchment is currently not offered by commercial tools that support the B method. More generally, tool support for retrenchment as not yet reached the same level of maturity as refinement.

This paper presents a refinement pattern to accommodate operation signatures that is fully compatible with the B method. An abstract instance of this refinement has been developed and verified with Atelier B [12]. The paper also shows how the pattern can be applied in a software development project where different execution platforms are employed (namely Java and Java Card). This instance has also been mechanically proved correct.

Future work include:

1. Proof that the constraints on the interface (or a weaker version thereof), listed as properties in the component $Context_I$, are necessary conditions to establish the refinement.
2. Automation of the proposed refinement pattern in existing tools supporting the B method [4] (this includes generating verification conditions based on the properties of Figure 7 instead of the more complex verification conditions for a generic refinement).

Both lines of work require the construction of an embedding of the B method in a proof system such as Isabelle [18], using an approach similar to that of HOL-Z [19]. Such embedding is necessary to obtain verified results on the B method (instead of its artifacts as we have done in this paper).

References

1. Chen, Z.: Java Card Technology for Smart Cards: Architecture and Programmer's Guide. Addison Wesley, Boston (2000)
2. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
3. Déharbe, D., Gomes, B.E.G., Moreira, A.M.: Automation of Java Card component development using the B method. In: 11th IEEE International Conference on Engineering of Complex Computer Systems, IEEE Computer Society (2006) 259–268
4. Gomes, B., Moreira, A., Déharbe, D., Moraes, K.: A ferramenta BSmart para o desenvolvimento rigoroso de aplicações Java Card com o método formal B. In: XIV Sessão de Ferramentas do XXI Simpósio Brasileiro de Engenharia de Software. (2007) 11–17
5. Banach, R., Poppleton, M.: Retrenchment: An engineering variation on refinement. In: B'98: Proceedings of the Second International B Conference, London, UK, Springer-Verlag (1998) 129–147
6. Hayes, I., Sanders, J.: Specification by interface separation. Formal Aspects of Computing **7** (1995) 430–439
7. Mikhajlova, A., Sekerinksi, E.: Class refinement and interface refinement in object-oriented programs. In: FME-97. Number 1313 in LNCS, Springer (1997) 82–101
8. Stepney, S., Cooper, D., Woodcock, J.: More powerful Z data refinement: Pushing the state of the art in industrial refinement. In: ZUM-98. Number 1493 in LNCS, Springer (1998)
9. Boiten, E., Derrick, J.: IO-refinement in Z. In: Third BCS-FACS Northern Formal Method Workshop. (1998)
10. Colin, S., Lanoix, A., Souquires, J.: Trustworthy interface compliancy: Data model adaptation using B refinement. Electronic Notes in Theoretical Computer Science **203**(7) (2009) 23 – 35
11. Clearsy: B language reference manual: version 1.8.6. `http://www.atelierb.eu` (2007)
12. ClearSy: Atelier B User Manual Version 4.0. `http://www.atelierb.eu`.
13. B-Core Ltd: The B-Toolkit. `http://www.b-core.com/btoolkit.html` last access: april 2009.
14. Fraser, S., Banach, R.: Configurable proof obligations in the Frog toolkit. In: Software Engineering and Formal Methods, London, UK, IEEE Computer Society (2007) 361–370
15. Fraser, S.: Mechanized Support for Retrenchment. PhD in computer science, Faculty of Engineering and Physical Sciences of University of Manchester, Manchester, UK (2008)
16. Spivey, J.: The Z Notation: a Reference Manual. 2nd edn. Prentice-Hall International Series in Computer Science. Prentice Hall (1992)
17. Lecomte, T., Méry, D., Cansell, D.: Patrons de conception prouvés. Journées Neptune (2007) ANR-006-SETIN-015.

18. Tobias Nipkow, Lawrence C. Paulson, M.W.: Isabelle/HOL A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2005)
19. Brucker, A.D., Rittinger, F., Wolff, B.: Hol-z 2.0: A proof environment for z-specifications. Journal of Universal Computer Science **9**(2) (2003) 152–172