Trustworthy interface compliancy: data model adaptation using B refinement

Samuel Colin, Arnaud Lanoix, Jeanine Souquières

Trustworthy interface compliancy: data model adaptation using B refinement

Samuel Colin, Arnaud Lanoix, Jeanine Souquières

LORIA – Université Nancy 2
Campus Scientifique, BP 239
F-54506 Vandœuvre lès Nancy cedex
{Samuel.Colin,Arnaud.Lanoix,Jeanine.Souquieres}@loria.fr

Abstract

In component-based software development approaches, components are considered as black boxes, communicating through required and provided interfaces which describe their visible behaviors. Each component interface is equipped with a suitable data model defining all the types occurring in the interface operations. The provided interfaces are checked to be compatible with the corresponding required interfaces, by the way of adapters. We propose a method to develop and verify these adapters when the interface data models are different, using the formal method B. The use of B assembling and refinement mechanisms eases the verification of the interoperability between interfaces and the correctness of the component assembly.

keywords: Component-based approach, correctness, interoperability, formal method, adapter, data model, interface.

1 Introduction

Component orientation is a new paradigm for the development of software-based systems. The basic idea is to assemble the software by combining pre-fabricated parts called software COTS (Commercial Off-The-Shelf) components, instead of developing it from scratch [22]. This procedure is similar to the construction methods applied in other engineering disciplines, such as electrical or mechanical engineering.

Software components are put together by connecting their interfaces. A *provided* interface of one component can be connected with a *required* interface of another component if the former offers the services needed to implement the latter. Hence, an appropriate description of the interfaces of a software component is crucial. In earlier papers [5, 4, 9] we have investigated how to formally specify interfaces of software components and how to prove their interoperability, using the formal method B, as presented in Section 2. Each component interface is equipped with a suitable data model defining all the types occurring in the signatures of interface operations.

In this paper, we study how to connect components with different data models by using adapters. We propose a method in three steps, sketched in Section 3, to build a trustworthy adapter following a refinement process: we start with the required interface and refine it until we can include the provided one. Each step expresses a level of interoperability, is supported by the prover and help us to establish the correctness of the adaptation. We support the presentation of this method with an example of an embedded system in Section 4. The paper finishes with the discussion of related work in section 5 and concluding remarks in section 6.

2 Using B for component-based development

We briefly describe the formal method B and explain how we use it in the context of component-based software. The architecture is modeled by UML diagrams (the components) annotated with B models associated to their interfaces. The B models are then used to verify the interface compliancy.

2.1 The B method

B is a formal software development method based on set theory, which supports an incremental development process using refinement [1]. Starting out from a textual description, a development begins with the definition of an abstract model, which can be refined step by step until an implementation is reached. Model refinement is a key feature for incrementally developing more and more detailed models, preserving correctness in each step. Each model consists in variables representing the state, operations representing the possible evolutions of this state and an invariant specifying the safety requirements.

The B method has been successfully applied in the development of several complex real-life applications, such as the METEOR project [2]. It is one of the few formal methods which has robust and commercially available support tools for the entire development life-cycle, from specification down to code generation [3]. It provides structuring primitives that allow one to compose models in various ways. Proofs of invariant consistency and refinement are part of each development and POs (Proof Obligations) are generated automatically by support tools such as AtelierB [21] or B4free [6]. Checking POs with B support tools is an efficient and practical way to detect errors introduced during development and to validate the B models.

2.2 Specifying component architectures

We define component-based systems using UML 2.0 composite structure diagrams [16]. They express the overall architecture of the system in terms of components and their required and provided interfaces. UML 2.0 Class diagrams express interface data models with their different attributes and methods.

Component interfaces are then specified as B models, which increases confidence in the developed systems: the correctness of the specifications, as well as the correctness of the refinement process can be checked with support tools. In an integrated development process, the B models can be obtained by applying systematic derivation rules from UML to B [14, 12].

2.3 Proving interoperability of component interfaces

The components must be connected in an appropriate way. To guarantee interoperability of components, we must consider each connection of a provided and a required interface contained in a software architecture and try to show that the interfaces are compatible. Using the B method, we prove that the B model of the provided interface is a correct B refinement of the required one. This result states that the provided interface constitutes a viable implementation of the required interface, and consequently that the two components are compliant as intended [4].

Often, to build a working component architecture, adapters need to be defined, connecting the required interfaces to the provided ones. An adapter is a piece of code that expresses the mapping between a required and a provided interface, usually a mapping between their variables at signature level. In [15], we have studied and proved an adapter specification defined in terms of a B refinement of the required interface that includes the B model of the provided (previously incompatible) interface.

2.4 An example of architecture

We illustrate our method with the case study of an embedded system where different sensors send alarm events. These alarms can be canceled by a control console and are memorized by a centralized database. The software architecture of this system is shown Figure 1 using the syntax of composite structure diagrams. It uses three COTS components:

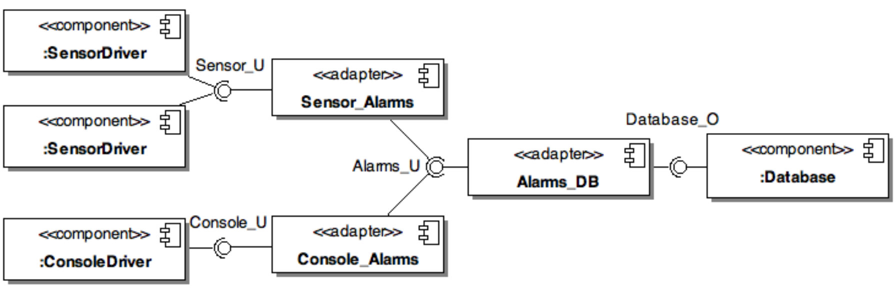
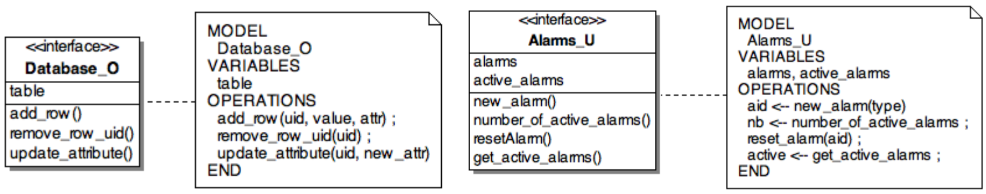


Figure 1: Component architecture

- Database provides database functionalities described by its provided interface Database_O as presented Figure 2 by UML diagrams and its associated B model (with only its signature). The B model of this interface with its data model and one of the operations is given Listing 1: (i) the types, represented as sets in B, used in the interface, (ii) variables as far as necessary to express the effects of the operations, (iii) an invariant on these variables and (iv) an operation specification.

- SensorDriver, the software part of each sensor, requires an interface Sensor_U to signal warning and error alarms to the system. These alarms need to be saved in the database. This component is used twice.

- ConsoleDriver, in charge to drive an alarm control console, requires an interface Console_U in order to query and cancel the alarms saved in the database.



(a) Interface Database_O provided by Database

(b) Interface Alarms_U

Figure 2: The interfaces and their associated B models

The interface Alarms_U, described in Figure 2 and Listing 2, expresses the global requirement of the alarms shared between the sensors and the console. Listing 3 presents the types used in Alarms_U.

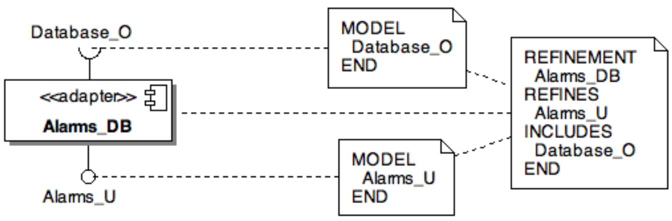


Figure 3: Adapter Alarms_DB

To assemble these three COTS, three adapters have been introduced:

- Alarms_DB maps the provided interface Database_O to the interface Alarms_U that shares the global resources (see Figures 1 and 3).

- Console_Alarms and Sensor_Alarms provide the required interface of each driver component using the interface Alarms_U.

```
MODEL Database_0
SETS
  Indices = {Uid, Value, Attribute}
VARIABLES
  table
INVARIANT
  table ∈ Indices → (ℕ1 ⇸ ℕ) ∧
  dom(table(Uid)) = dom(table(Value)) ∧
  dom(table(Uid)) = dom(table(Attribute)) ∧
  table(Uid) ∈ (ℕ1 ⤔ ℕ)
INITIALISATION
  table := { Uid ↦ ∅, Value ↦ ∅, Attribute ↦ ∅ }
OPERATIONS

  add_row(uid, value, attr)=
    PRE
      uid ∈ ℕ ∧
      value ∈ ℕ ∧
      attr ∈ ℕ ∧
      ∀ii .(( ii ∈ dom(table(Uid))) ⇒ (uid ≠ table(Uid)( ii )))
    THEN
      ANY indice
      WHERE indice ∈ ℕ1 − dom(table(Uid))
      THEN
        table := table ⩤ { Uid ↦ (table(Uid) ⩤ {indice ↦ uid}),
                           Value ↦ (table(Value) ⩤ {indice ↦ value}),
                           Attribute ↦ (table(Attribute) ⩤ {indice ↦ attr})}
      END
    END;

  remove_row_uid(uid) =
    PRE
      uid ∈ ran(table(Uid))
    THEN
      ANY indice
      WHERE indice ∈ dom(table(Uid)) ∧ table(Uid)(indice) = uid
      THEN
        table := table ⩤ { Uid ↦ ( (dom(table(Uid)) − {indice}) ◁ table(Uid)),
                           Value ↦ ( (dom(table(Value)) − {indice}) ◁ table(Value)),
                           Attribute ↦ ( (dom(table(Attribute)) − {indice}) ◁ table(Attribute)) }
      END
    END;

  update_attribute(uid, new_attr) =
    PRE
      uid ∈ ran(table(Uid)) ∧
      new_attr ∈ ℕ
    THEN
      ANY indice
      WHERE indice ∈ dom(table(Uid)) ∧ table(Uid)(indice) = uid
      THEN
        table := table ⩤ { Attribute ↦ (table(Attribute) ⩤ {indice ↦ new_attr}) }
      END
    END
END
```

Listing 1: B model of Database_O

```
MODEL Alarms_U
SEES Types
VARIABLES
  alarms,  active_alarms
INVARIANT
  alarms ⊆ AlarmIds ∧
  active_alarms ⊆ alarms
INITIALISATION
  alarms := ∅ ||
  active_alarms := ∅
OPERATIONS

  nb ⟵ number_of_active_alarms =
  BEGIN
    nb := card( active_alarms )
  END;

  active ⟵ get_active_alarms =
  BEGIN
    active := active_alarms
  END;

  reset_alarm (aid) =
  PRE aid ∈ active_alarms
  THEN
    active_alarms := active_alarms − { aid }
  END;

  aid ⟵ new_alarm(type) =
  PRE
    type ∈ AlarmTypes
  THEN
    ANY uid
    WHERE uid ∈ AlarmIds − alarms
    THEN
      aid := uid ||
      alarms := alarms ∪ {uid} ||
      active_alarms := active_alarms ∪ {uid}
    END
  END

END
```

Listing 2: B model of the interface Alarms_U

```
MODEL Types
SETS
  DeviceIds;
  AlarmIds;
  AlarmTypes;
  AlarmStatus = {Inactive, Active}
END
```

Listing 3: The types used in the development

In the rest of this paper, we focus on the development and the correctness of the adapter Alarms_DB which must provide Alarms_U using Database_O. In terms of B models, we have to prove that Alarms_DB is a refinement of Alarms_U including Database_O in a similar way to [15], as shown Figure 3.

3 Trustworthy method to adapt interface data models

Let I_U be an interface required by a component A and I_O an interface provided by a component B. Our goal is to develop an adapter that implements the data model of I_U using the data model of I_O. In other words, the adapter must express I_U in terms of the variables, data types and operations of I_O.

I_U and I_O are defined by B models as presented Figure 4. We denote by V_U and V_O their sets of variables and by OP_U and OP_O their sets of operations, respectively. We note D_U (resp. D_O) the set of data types of the variables V_U (resp. V_O).

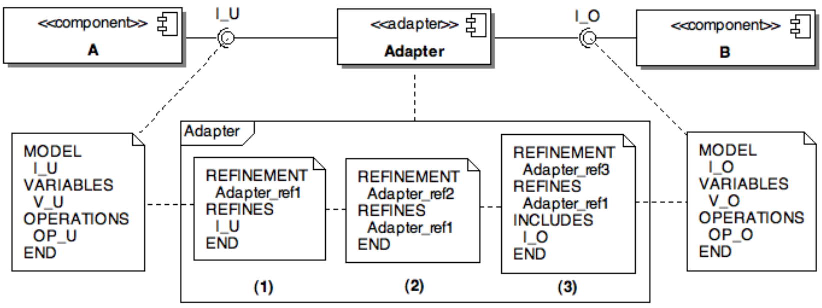


Figure 4: Process of the adapter development

The adapter must be trustworthy and the proof of the adaptation becomes complex when data models of I_U and I_O are different. In order to ease this proof, we develop the adapter by incremental refinements guided by the transformation of the variables of I_U into the variables of I_O.

3.1 Process description

The adaptation process is guided by the interface I_O and consists of three refinement steps. Each step is proved by using the B refinement mechanism.

(1) Variables adaptation

This step prepares a matching between the variables of I_U and I_O:

- each variable of V_U is transformed into a new variable of V_U', "corresponding to" a variable of V_O, using the data types D_U,
- the body of each operation of OP_U is transformed with respect to these new variables into OP_U'.

(2) Data types adaptation

This step provides a matching between the data types of I_U and I_O:

- each variable of V_U' expressed on D_U is transformed into a new variable of V_U" expressed using the data types D_O. To do that, typecasting functions between D_U and D_O (and reciprocally) have to be defined,
- the body of each operation of OP_U' is transformed with respect to the new variables V_U" into OP_U".

(3) Provided interface inclusion

This step, which has been prepared by the two previous ones, consists in:

- associating each variable of V_U" to V_O variables,
- expressing each operation of OP_U" in terms of operations of OP_O.

3.2 B as a guideline for the adaptation steps

When the required and the provided interfaces are defined on the same data types, the adaptation becomes a problem of transforming variables and calling the right operations. When the interfaces are similar modulo their data types, the problem is reduced to find whether the elements of D_U are subtypes of elements of D_O, and then calling the operations with the transformed variables. In the latter case, the role of the adapter is simply the role of a variable wrapper.

With the use of B, the adaptation process and therefore the adapter itself, is validated by the proof of the different refinement steps. A direct consequence is that the adaptation process is less guided by the intuition of the developer and more by mathematical and logical laws. Hence each step of the process might require several refinement steps in practice in order to provably guarantee that the transformation is correct. As a matter of fact, the B refinement mechanism encourages this practice.

Furthermore, in some transformation steps, functions are introduced as constants, which need to be explicit in the implementation step. Hence our method is no silver bullet: great care has to be taken when these functions appear. The developer of the adapter *has to* ensure that the transformation functions exist. Their existence can be more easily stated if the refinement steps are limited to simple, intuitive and progressive transformations. For instance, instead of transforming enumerated values of a set directly to the set of natural numbers, it is wiser to first transform it to a set of numbers modulo the number of enumerated values and then transform it to the full set of natural numbers. This way the proof of the refinements become easier.

4 Case study

We now show the application of this method to develop and prove the adapter Alarms_DB that must provide the interface Alarms_U using the interface Database_O, as presented

Figure 5. The specification of the B operations (not shown in this figure) is modified according to the variable transformations realized at each step of the development[1].

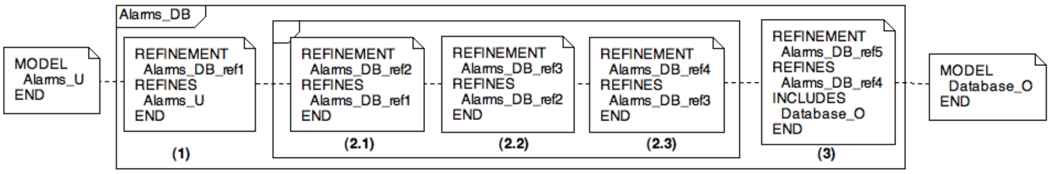


Figure 5: Refinement steps of the adapter Alarms_DB

4.1 Variables adaptation

The first step consists in adapting the variables alarms and active_alarms of the interface data model of Alarms_U to the interface data model of Database_O. During this step, we do not introduce new data types. In the database, each entry in the table is characterized by an identifier Uid which has a corresponding Value and an Attribute. Guided by these three variables, we consider mapping the alarms with the Uid field, the type of an alarm with the Value field and its activity status (active_alarms) with the Attribute field.

We introduce three new variables corresponding to Uid, Value and Attribute: alarms_ids is directly associated to alarms, whereas AlarmTypes and AlarmStatus are functions expressing the type and the status of an alarm as illustrated Listing 4. The proof of this refinement consists of 18 POs, among which 4 have been proved interactively.

4.2 Data types adaptation

Typecasting is a frequent source of bugs, as limit conditions are often overlooked. Consequently, the second step might possibly be the harder one: great care must be taken when casting the variables from one type to another one. The proof process exhibits these limit conditions and oblige to check their validity. In our adaptation process, the typecasting functions are introduced as constants. It means that the validity of the adaptation relies on the existence of these functions, hence it is wiser to choose typecasting functions with well-understood mathematical properties. To ease the proof verification, we break down the data types adaptation step into three refinements:

(2.1) typecasting the non-functional variables (alarms_ids),

(2.2) typecasting the domain (in the mathematical sense) of each functional variable (alarms_type and alarms_status),

(2.3) typecasting the codomain of each functional variable (the already transformed alarms_type and alarms_status).

[1]Complete B models are published in [7].

```
REFINEMENT Alarms_DB_ref1
REFINES Alarms_U
SEES Types
VARIABLES
  alarms_ids , alarms_status , alarms_type
INVARIANT
  alarms_ids = alarms ∧
  alarms_status ∈ alarms_ids → AlarmStatus ∧
  alarms_type ∈ alarms_ids → AlarmTypes ∧
  alarms_status = active_alarms×{Active} ∪ (alarms_ids − active_alarms)×{Inactive}
ASSERTIONS
  ({Active}∗active_alarms ∪ {Inactive}∗(alarms−active_alarms))[{Active}] = active_alarms
INITIALISATION
   alarms_ids := ∅ ||
   alarms_status := ∅×AlarmStatus ||
   alarms_type := ∅×AlarmTypes
OPERATIONS

  nb ←− number_of_active_alarms =
  BEGIN
    nb := card( alarms_status ⁻¹[{Active}])
  END;

  active ←− get_active_alarms =
  BEGIN
    active := alarms_status ⁻¹[{Active}]
  END;

  reset_alarm (aid) =
  BEGIN
    alarms_status := alarms_status <+ { aid ↦ Inactive }
  END;

  aid ←− new_alarm(type) =
  ANY uid
  WHERE uid ∈ AlarmIds − alarms_ids
  THEN
    aid := uid ||
    alarms_ids := alarms_ids ∪ {uid} ||
    alarms_type := alarms_type <+ { uid ↦ type } ||
    alarms_status := alarms_status <+ { uid ↦ Active }
  END

END
```

Listing 4: Step **(1)** of the adaptation process

4.2.1 Typecasting the non-functional variables

The alarms_ids variable will be represented at the end of the process by the Uid field of the database. We introduce a constant function id_cast in order to typecast from AlarmIds to the natural numbers, i.e. the type of the Uid field. We therefore represent the alarms_ids by a new variable nat_ids and we add a relationship between both variables in the invariant. The other variables are unchanged, and the result is shown in Listing 5. The invariant expresses the fact that nat_ids is the image of the alarms_ids by id_cast. The proof of this refinement consists of 8 POs, among which 2 have been proved interactively.

```
REFINEMENT Alarms_DB_ref2
REFINES Alarms_DB_ref1
SEES Types
CONSTANTS
  id_cast
PROPERTIES
  id_cast ∈ AlarmIds ⤖ ℕ
VARIABLES
  nat_ids , alarms_status , alarms_type
INVARIANT
  nat_ids = id_cast [ alarms_ids ]
ASSERTIONS
  ∀aid .(( aid ∈ alarms_ids ) ⇒ ( id_cast ( aid ) ∈ id_cast [dom(alarms_type)]) )
INITIALISATION
  nat_ids := ∅ ||
  alarms_status := ∅×AlarmStatus ||
  alarms_type := ∅×AlarmTypes
OPERATIONS

  aid ⟵ new_alarm(type) =
  ANY uid_nat
  WHERE
    uid_nat ∈ ℕ ∧
    uid_nat /∈ nat_ids
  THEN
    aid := id_cast ⁻¹(uid_nat) ||
    nat_ids := nat_ids ∪ {uid_nat} ||
    alarms_type := alarms_type ⩤ { id_cast ⁻¹(uid_nat) ↦ type } ||
    alarms_status := alarms_status ⩤ { id_cast ⁻¹(uid_nat) ↦ Active }
  END

END
```

Listing 5: Step **(2.1)** of the adaptation process

4.2.2 Typecasting the domain of each functional variable

The variables alarms_status and alarms_type depend on alarms_ids. As alarms_ids has been transformed into nat_ids, we must also transform alarms_status and alarms_type so that they depend rather on nat_ids. We thus replace them by the variables nat_status and nat_type. The result is presented in Listing 6. The invariant helps relating nat_status with nat_ids, i.e. it states that nat_status is the composition of the functions alarm_status and id_cast. The proof of this refinement consists of 14 POs, among which 5 have been proved interactively.

```
REFINEMENT Alarms_DB_ref3
REFINES Alarms_DB_ref2
SEES Types
VARIABLES
 nat_ids , nat_status , nat_type
INVARIANT
 nat_status ∈ nat_ids → AlarmStatus ∧
 nat_type ∈ nat_ids → AlarmTypes ∧
 nat_status ⁻¹ = (alarms_status⁻¹; id_cast)
INITIALISATION
   nat_ids := ∅ ||
   nat_status := ∅ ||
   nat_type := ∅
OPERATIONS

  nb ⟵ number_of_active_alarms =
  BEGIN
    nb := card( nat_status ⁻¹[{Active}])
  END;

  active ⟵ get_active_alarms =
  BEGIN
    active := id_cast ⁻¹[ nat_status⁻¹[{Active}] ]
  END;

  reset_alarm ( aid ) =
  BEGIN
    nat_status := nat_status ⩤ { id_cast ( aid ) ↦ Inactive }
  END;

  aid ⟵ new_alarm(type) =
  ANY uid_nat
  WHERE
    uid_nat ∈ ℕ ∧
    uid_nat /∈ nat_ids
  THEN
    aid := id_cast ⁻¹(uid_nat) ||
    nat_ids := nat_ids ∪ {uid_nat} ||
    nat_type := nat_type ⩤ { uid_nat ↦ type } ||
    nat_status := nat_status ⩤ { uid_nat ↦ Active }
  END

END
```

Listing 6: Step **(2.2)** of the adaptation process

4.2.3 Typecasting the codomain of each functional variable

Before this step, the codomains of nat_status and nat_type are not in the data types of Database_O. We need to typecast these codomains, namely AlarmStatus and AlarmTypes, to the corresponding data types of the fields of the database, i.e. Attribute and Value respectively. These fields contain natural numbers, hence we introduce two constant functions named status_cast and type_cast which map AlarmStatus and AlarmTypes to natural numbers.

```
REFINEMENT Alarms_DB_ref4
REFINES Alarms_DB_ref3
SEES Types
CONSTANTS
 type_cast , status_cast
PROPERTIES
 type_cast ∈ AlarmTypes ⤖ 1..card(AlarmTypes) ∧
 status_cast ∈ AlarmStatus ⤖ 1..card(AlarmStatus)
CONCRETE_VARIABLES
 uid_gen
VARIABLES
 ids_nn , status_nn , type_nn
INVARIANT
 uid_gen ∈ ℕ ∧
 ids_nn = nat_ids ∧
 status_nn ∈ nat_ids → 1.. card(AlarmStatus) ∧
 type_nn ∈ nat_ids → 1.. card(AlarmTypes) ∧
 uid_gen > max(nat_ids) ∧
 status_nn = (nat_status ; status_cast ) ∧
 type_nn = (nat_type; type_cast )
ASSERTIONS
 status_cast ⁻¹[status_cast[{Active }]] = {Active}
INITIALISATION
  uid_gen := 0 ||
  ids_nn := ∅ ||
  status_nn := ∅ ||
  type_nn := ∅
OPERATIONS

  nb ⟵ number_of_active_alarms =
  BEGIN
    nb := card( status_nn⁻¹[status_cast[{Active }]])
  END;

  active ⟵ get_active_alarms =
  BEGIN
    active := id_cast ⁻¹[ status_nn⁻¹[status_cast[{Active}]] ]
  END;

  reset_alarm ( aid ) =
  BEGIN
    status_nn := status_nn ⩤ { id_cast ( aid ) ↦ status_cast( Inactive ) }
  END;

  aid ⟵ new_alarm(type) =
  BEGIN
    aid := id_cast ⁻¹(uid_gen) ||
    ids_nn := ids_nn ∪ {uid_gen} ||
    type_nn := type_nn ⩤ { uid_gen ↦ type_cast(type) } ||
    status_nn := status_nn ⩤ { uid_gen ↦ status_cast(Active) } ||
    uid_gen := uid_gen + 1
  END

END
```

Listing 7: Step **(2.3)** of the adaptation process

The variables status_nn and type_nn that we have introduced correspond to nat_status and nat_type respectively. As the codomains of status_nn and type_nn are the natural numbers, the codomains of nat_status and nat_type are transformed by the typecasting functions mentioned above. For notation consistency, we rename nat_ids into ids_nn. Moreover, we introduce a new variable uid_gen for producing a new unique index each time a new alarm is added in the database. All these transformations are shown in Listing 7. The proof of this refinement consists of 20 POs, among which 6 have been proved interactively.

Note that with this last invariant, we obtain that alarm_status can be replaced by all the constants and variables we introduced along the refinements.
We have: $\mathsf{alarm\_status} = \mathsf{status\_cast}^{-1} \circ \mathsf{status\_nn} \circ \mathsf{id\_cast}$. The functions $\mathsf{status\_nn} \circ \mathsf{id\_cast}$ and $\mathsf{status\_cast} \circ \mathsf{alarm\_status}$ commute. This property is illustrated by Figure 6.

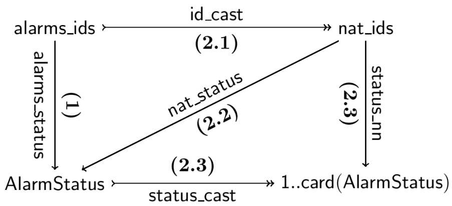


Figure 6: Commutation diagram

4.3 Provided interface inclusion

In the last step, we establish the relationships between the ids_nn, status_nn and type_nn variables and the fields Uid, Attribute and Value of table as illustrated in Listing 8. We also perform the operation calls to Database_O to express the operations of Alarms_U: the body of the operation new_alarm consists mainly of a call to the operation add_row of Database_O. The proof of this refinement consists of 19 POs, among which 5 have been proved interactively.

The proof of this last step is at the crossroad of the POs of the refinements and the POs of the included (provided) interface, hence the POs here tend to be unreadable because of the size of the terms. Fortunately, the shape of the formulas also tend to resemble the POs of the refinements and the POs of Database_O. Hence most of the time similar strategies with the proof strategies of the refinements and the included interface can be used for proving the last step.

The proof process for the development of this example, including the proofs of the consistency of the B models of the interfaces (Listings 1 and 2) and the proofs of the different refinement steps (Listings 4, 5, 6, 7 and 7), is composed of 108 POs, among which 30 POs have been proved interactively (see Table 1 for details).

	Obvious POs	*POs*	*Interactive POs*
Database_O	3	24	8
Alarms_U	11	5	0
Alarms_DB_ref1	26	18	4
Alarms_DB_ref2	21	8	2
Alarms_DB_ref3	25	14	5
Alarms_DB_ref4	39	20	6
Alarms_DB_ref5	23	19	5
TOTAL	**148**	**108**	**30**

Table 1:

```
REFINEMENT Alarms_DB_ref5
REFINES Alarms_DB_ref4
SEES Types
INCLUDES Database_O
INVARIANT
 table(Uid)[dom(table(Uid))] = ids_nn ∧
 (table(Uid)⁻¹;table(Attribute)) = status_nn ∧
 (table(Uid)⁻¹;table(Value)) = type_nn
INITIALISATION
 uid_gen := 0
OPERATIONS

  nb ⟵ number_of_active_alarms =
  BEGIN
    nb := card(table(Uid) [table(Attribute)⁻¹[status_cast[{Active}]] ])
  END;

  active ⟵ get_active_alarms =
  BEGIN
    active := id_cast⁻¹[table(Uid)[ (table(Attribute))⁻¹[status_cast[{Active}]] ]]
  END;

  reset_alarm(aid) =
  BEGIN
    update_attribute(id_cast(aid), status_cast(Inactive))
  END;

  aid ⟵ new_alarm(type) =
  BEGIN
    aid := id_cast⁻¹(uid_gen) ||
    uid_gen := uid_gen + 1 ||
    add_row(uid_gen, type_cast(type), status_cast(Active))
  END

END
```

Listing 8: Step **(3)** of the adaptation process

5 Related work

One of the first approaches of module reuse through interface adaptation is the approach of Purtilo and Atlee [17]: they use a dedicated language (called Nimble) for relating a required interface to a provided one, where the adaptation is made by the developer. Our approach is similar modulo the formalism used for representing the interfaces: instead of a dedicated language, we use UML and the B method. We have the benefit of relying on standards. Furthermore we overcome the limited semantics of their approach because we use a formal tool for expressing and verifying the interface adaptation.

Dynamic component adaptation [13, 10] goes further than our approach by proposing methods for adapting *at run-time* components by finding suitable adapter components based on the interfaces of the components to adapt. Unfortunately these methods have strong requirements (knowing inheritance relationships, runtime mapping of interface relationships, ...) and rely primarily on types and/or object-oriented peculiarities, hence they are limited to subtype-like adaptations. This is not possible with our approach because trustworthiness would require also proving these strong requirements at run-time.

Our method allows nevertheless a broader range of possible adaptations (not limited to subtypes of a provided interface).

The paper [8] presents a framework for modeling component architectures using formal techniques (Petri Net and CSP): connections between required and provided interfaces (called import and export interfaces) of components are represented by graph transformations (composition, embedding, extension and refinement). Our approach is similar. We use B formal method to express transformations as refinement between the required interface and the provided one.

Zaremski and Wing [23] propose an interesting approach to compare two software components. It is determined whether one component can be substituted for another. They use formal specifications to model the behavior of components and the Larch prover to prove the specification matching of components.

Reussner et al. [18, 19] present adapters in the context of concurrent systems. They consider only a certain class of protocol interoperability problems and generate adapters for bridging component protocol incompatibilities, using interface described by finite parameterized state machines.

The refinement steps of our approach for building an adapter can also be viewed as steps for building morphisms between interfaces. Such methods, for instance the methods presented by Smith [20], are based on signature algebras and theory category. Our approach is rather practical because we choose the B method for expressing the interfaces. The B method is indeed easier for software engineers to understand because it is based on set theory. Our results resemble much with interface morphisms, thus these methods could provide means for automating our approach better.

6 Conclusion

The component-based paradigm has received considerable attention in the software development field in industry and academia like in other engineering domains. In this approach, components are considered as black-boxes described by their visible behavior and their required and provided interfaces. To construct a working system out of existing components, adapters are introduced. An adapter is a piece of glue code that realizes the required interface using the provided interfaces. It expresses the mapping between required and provided variables and how required operations are implemented in terms of the provided ones. We have presented a method in three steps to adapt complex data models, each step expressing a level of interoperability and establishing the correctness of the adaptation.

Using the formal method B and its refinement and assembling mechanisms to model the component interfaces and the adapters, we pay special attention to the question of guaranteeing the interoperability between the different components. The B prover guarantees that the adapter is a correct implementation of the required functionalities in terms of the existing components. With this approach, the verification of the interoperability between the connected components is achieved at the signature, the semantic and the protocol levels.

We are currently working on a method for adding dependability features to component-based software systems. The method is applicable if the dependability features add new behavior to the system, but do not change its basic functionality [11]. The idea is to start with a software architecture whose central component is an application component that implements the behavior of the system in the normal case. The application component is connected to other components, possibly through adapters. It is then possible to enhance the system by adding dependability features in such a way that the central application component remains untouched. Adding dependability features necessitates to evolve the overall system architecture by replacing or newly introducing hardware or software components. The adapters contained in the initial software architecture have to be modified, whereas the other software components need not to be changed. Thus, the dependability of a component-based system can be enhanced in an incremental way.

References

[1] J.-R. Abrial. *The B Book*. Cambridge University Press, 1996.

[2] P. Behm, P. Benoit, and J.M. Meynadier. METEOR: A Successful Application of B in a Large Project. In *Integrated Formal Methods, IFM99*, volume 1708 of *LNCS*, pages 369–387. Springer Verlag, 1999.

[3] D. Bert, S. Boulmé, M-L. Potet, A. Requet, and L. Voisin. Adaptable Translator of B Specifications to Embedded C Programs. In *Integrated Formal Method, IFM'03*, volume 2805 of *LNCS*, pages 94–113. Springer Verlag, 2003.

[4] S. Chouali, M. Heisel, and J. Souquières. Proving Component Interoperability with B Refinement. *Electronic Notes in Theoretical Computer Science*, 160:157–172, 2006.

[5] S. Chouali and J. Souquières. Verifying the compatibility of component interfaces using the B formal method. In CSREA Press, editor, *International Conference on Software Engineering Research and Practice (SERP'05)*, pages 850–856, 2005.

[6] Clearsy. B4free. Available at *http://www.b4free.com*, 2004.

[7] S. Colin, A. Lanoix, and J. Souquières. Trustworthy interface compliancy: data model adaptation. Research Report hal-00123884, LORIA, Jan 2007. http://hal.archives-ouvertes.fr/hal-00123884.

[8] H. Ehrig, J. Padberg, B. Braatz, M. Klein, F. Orejas, S. Perez, and E. Pino. A generic framework for connector architectures based on components and transformation. In *FESCA'04, satellite of ETAPS'04*, number 108, pages 53–67. ENTCS, 2004.

[9] D. Hatebur, M. Heisel, and J. Souquières. A Method for Component-Based Software and System Development. In IEEE Computer Society, editor, *Proceedings of the 32tnd Euromicro Conference on Software Engineering And Advanced Applications*, pages 72–80, 2006.

[10] G. Kniesel. Type-safe delegation for run-time component adaptation. *Lecture Notes in Computer Science*, 1628:351–366, 1999.

[11] A. Lanoix, D. Hatebur, M. Heisel, and J. Souquières. Enhancing Dependability of Component-based Systems. Research Report hal-00123999, LORIA, Dec 2006. http://hal.archives-ouvertes.fr/hal-00123999.

[12] H. Ledang and J. Souquières. Modeling class operations in B: application to UML behavioral diagrams. In *ASE'2001 : 16th IEEE International Conference on Automated Software Engineering*, pages 289–296. IEEE Computer Society, 2001.

[13] K.-U. Mätzel and P. Schnorf. Dynamic component adaptation. Technical report, Ubilab, Union Bank of Switzerland, Zürich, Switzerland, June 1997.

[14] E. Meyer and J. Souquières. A systematic approach to transform OMT diagrams to a B specification. In *Proceedings of the Formal Method Conference*, LNCS 1708, pages 875–895. Springer-Verlag, 1999.

[15] I. Mouakher, A. Lanoix, and J. Souquières. Component Adaptation: Specification and Verification. In *Proc. of the 11th Int. Workshop on Component Oriented Programming (WCOP 2006)*, pages 23–30, July 2006.

[16] Object Management Group (OMG). *UML Superstructure Specification*, 2005. version 2.0.

[17] J.M. Purtilo and J.M. Atlee. Module reuse by interface adaptation. *Software - Practice and Experience*, 21(6):539–556, 1991.

[18] R. H. Reussner and H. W. Schmidt. Using Parameterised Contracts to Predict Properties of Component Based Software Architectures. In Ivica Crnkovic, Stig Larsson, and Judith Stafford, editors, *Workshop On Component-Based Software Engineering (in association with 9th IEEE Conference and Workshops on Engineering of Computer-Based Systems), Lund, Sweden, 2002*, 2002.

[19] R. H. Reussner, H. W. Schmidt, and I. H. Poernomo. Reasoning on software architectures with contractually specified components. In A. Cechich, M. Piattini, and A. Vallecillo, editors, *Component-Based Software Quality: Methods and Techniques*. 2003.

[20] D. R. Smith. Constructing specification morphisms. *Journal of Symbolic Computation*, 15(5/6):571–606, 1993.

[21] Steria – Technologies de l'information. *Obligations de preuve: Manuel de référence, version 3.0*, 1998.

[22] C. Szyperski. *Component Software*. ACM Press, Addison-Wesley, 1999.

[23] A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Transaction on Software Engeniering Methodolology*, 6(4):333–369, 1997.