Trustworthy interface compliancy: data model adaptation

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 signature of 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.

1 Introduction

Component orientation is a new paradigm for the development of software-based systems. The basic idea is to assemble the software by combination of pre-fabricated parts called software components, instead of developing it from scratch. 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 it offers the services needed to implement the required interface. Hence, an appropriate description of the interfaces of a software component is crucial. In earlier papers [4,6,11], we have investigated how to formally specify interfaces of software components and how to demonstrate their interoperability, using the formal method B. Each component interface is equipped with a suitable data model defining all the types occurring in the signature of interface operations.

In this paper, we study how to connect components by the way of adapters when their interface data models are different. We propose a method in three steps to built 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.

The rest of the paper is organized as follow: in Section 2, we describe how we support component-based development using the formal method B. We then describe our method to connect components in case of mismatching interface data models in Section 3. The method is illustrated by the case study of an embedded system presented 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. We formally express provided and required interfaces using B models in order to verify their compatibility.

2.1 The formal method B

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. The refinement of models is a key feature for incrementally developing more and more detailed models, preserving correctness in each step.

The method B 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 invariance and refinement are part of each development. The proof obligations are generated automatically by support tools such as AtelierB [16] or B4free [5]. Checking proof obligations with B support tools is an efficient and practical way to detect errors introduced during development.

2.2 Specifying component architectures

We define component-based systems using UML 2.0 composite structure diagrams [13]. They express the overall architecture of the system in terms of components and their required and provided interfaces. UML 2.0 Class diagrams serve to 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 [10,9].

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 method B, we prove that the B model of the provided interface is a correct B refinement of the required one. This means that the provided interface constitutes an implementation of the required interface, and we conclude that the two components can be connected as intended [4].

Often, to construct a working component architecture, adapters have to be defined, connecting the required interfaces to the provided ones. An adapter is a piece of glue code that realizes the required interface using the provided interface. At the signature level, it expresses the mapping between required and provided variables. In [11], we have studied an adapter specification and its verification by giving a B refinement of the adaptation that refines the B model of the required interface including 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:

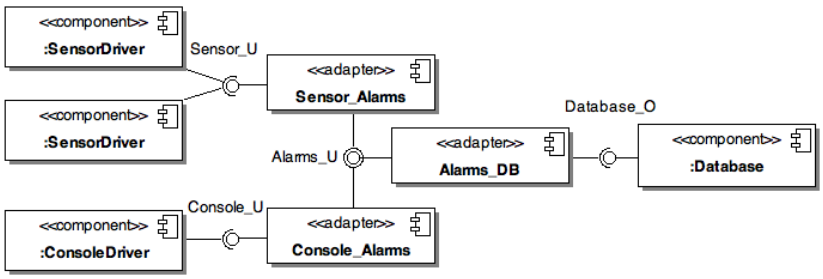
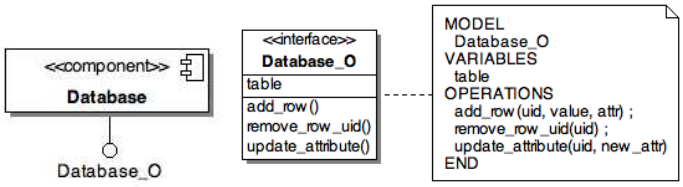


Figure 1. Component architecture

- The component 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 Figure 4 (listing 1): (i) the types 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.
- The component SensorDriver 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.
- The component 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) Database (b) Database_O

Figure 2. Component Database and its associated interface Database_O

The interface Alarms_U, described Figures 3 and 4 (listing 2), expresses the global requirement of the alarms shared between the sensors and the console. Listing 3 of Figure 4 presents the types used in Alarms_U.

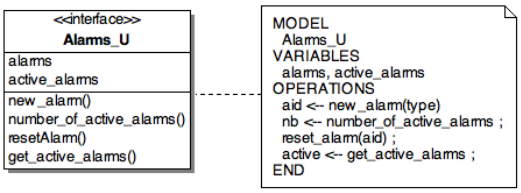


Figure 3. Interface Alarms_U

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)
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;

Listing 1. B model of Database_O

MODEL Alarms_U
SEES Types
VARIABLES
alarms, active_alarms
INVARIANT
alarms ⊆ AlarmIds ∧
active_alarms ⊆ alarms
OPERATIONS

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. Types

Figure 4. The component interfaces expressed with B

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

- The adapter Alarms_DB to map the provided interface Database_O that shares the global resources, to the interface Alarms_U (see Figures 1 and 5).

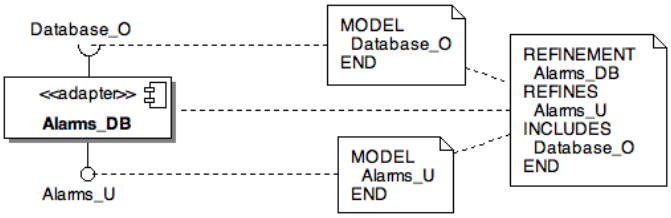


Figure 5. Adapter Alarms_DB

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

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 [11] as shown Figure 5.

3 Trustworthy method to adapt interface data models

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

I_U and I_O are defined by B models. We will refer to their variables V_U and V_O, and to their operations OP_U and OP_O respectively. We notice D_U (resp. D_O) the data types of the variables V_U (resp. V_O) as presented Figure 6.

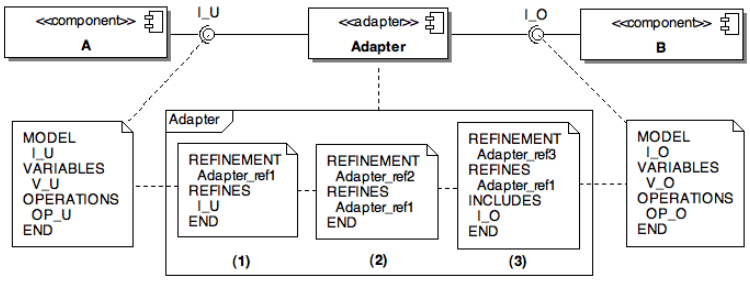


Figure 6. Process of the adapter development

The adapter must be trustworthy and its proof becomes complex when data models of I_U and I_O are different. To ease this proof, we develop the adapter by incremental refinements guided by the transformation of 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 provides 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 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, relations between D_U and D_O have to be defined ;
- the body of each operation OP_U' is transformed with respect to these new variables V_U'' into OP_U''.

(3) Provided interface inclusion

This step, which have 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 the call the relevant 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. Similarly, when the interfaces are similar modulo their data types, the problem is reduced to find whether D_U are subtypes of D_O, and then indeed calling the operations with the transformed variables. In that 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 developer's intuition and more by mathematical and logical laws. That means that 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 will need to made 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 *will have to* ensure that the translation 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 translate it to a set of numbers modulo the number of enumerated values and then translate it to the full set of natural numbers. The proof of the refinements become easier.

4 Case study

We now show the application of the previous developement process to develop and prove an adapter that provides the interface data model of Alarms_U using the interface data model of Database_O. Each time we will refer to a listing in this section, we implicitly refer to figure 7. The specification of the operations (not shown in this figure) is of course modified according to the transformations realized at each step [1].

4.1 Variables adaptation

The first step consists in adapting alarms and active_alarms of the interface data model of Alarms_U to the interface data model of Database_O. In 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 choose to associate the alarm identifiers (alarms_ids) with the Uid field, the type of an alarm (AlarmTypes) with the Value field and its activity status (AlarmStatus) with the Attribute field.

This matching between an alarm identifier and its attributes can be easily expressed by B functions. Hence we come up with a function for expressing the type of an alarm, and a second one for the status of an alarm as illustrated listing 4. The proof of this refinement consists of 18 formulas, whose 4 have beeen proved interactively.

4.2 Data types adaptation

The second step might possibly be the harder one: great care must be taken when casting the variables from one type to another one. This is because typecasting is a frequent source of bugs, as limit conditions are often overlooked. The proof process exhibits these conditions and oblige to check their validity. In our 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. We broke down the "data types adaptation" step into three refinements, characterized as follows:

- Typecasting the non-functional variables (alarms_ids)
- Typecasting the domain (in the mathematical sense) of each functional variable (alarms_type and alarms_status)
- Typecasting the codomain of each functional variable (the already transformed alarms_type and alarms_status).

4.2.1 Typecasting the non-functional variables

The alarms_ids variable will be represented in the end by the Uid field of the database. We thus introduce a constant 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

[1] Complete B models are published in []

```
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}
END
```

Listing 4. First step of the refinement

```
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 ]
END
```

Listing 5. Second step (2.1) of the refinement

```
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)
END
```

Listing 6. Second step (2.2) of the refinement

```
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 )
END
```

Listing 7. Second step (2.3) of the refinement

```
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
END
```

Listing 8. Third step of the refinement

Figure 7. Adapting Alarms_U to Database_O in five refinements

by the variable nat_ids and add this relationship between both variables in the gluing invariant. The other variables are unchanged, and the result is shown in listing 5. The proof of this refinement consists of 8 formulas, whose 2 have had been proved interactively.

4.2.2 Typecasting the domain of each functional variable

The alarms_status and alarms_type depend on alarms_ids. As alarms_ids has been transformed into nat_ids, we must transform alarms_status and alarms_type so that they depend rather on nat_ids. We thus replace them with their equivalent nat_status and nat_type. The result is presented in listing 6. The proof of this refinement consists of 14 formulas, whose 5 have been proved interactively.

4.2.3 Typecasting the codomain of each functional variable

At this step, the codomains of nat_status and nat_type do not appear in the data types of Database_O. We need to typecast these codomains, namely AlarmStatus and AlarmTypes, to the corresponding data types fields of the database, i.e. Attribute

and Value respectively. These fields contain natural numbers, hence the typecasting functions will map AlarmStatus and AlarmTypes to natural numbers. These functions are introduced as constants and named status_cast and type_cast.

We thus introduce the variables status_nn and type_nn which correspond to the 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 have been transformed by the typecasting functions mentioned above. For consistent notations, we rename nat_ids into ids_nn.

We finally introduce a new variable uid_gen for producing a new unique index each time a new alarm will be added in the database. All these transformations are shown listing 7. The proof of this refinement consists of 20 formulas, whose 6 have been proved interactively.

4.3 Provided interface inclusion

In the last step, we establish the relationship between status_nn and type_nn and the fields of the data base as illustrated listing 8. We also do the calls to the operations of the data base with the transformed data model. The proof of this refinement consists of 19 formulas, whose 5 havebeen proved interactively.

The proof of this last step is most difficult to perform because it relies on the proofs made along the refinement *and* the proofs of the Database_O abstract model. Our approach brought a benefit here: while the formulas are bigger in size because of the size of the terms, they have a shape similar with the proof obligations of the provided interface. Hence the proofs can be done by following roughly the proof steps for the provided interface.

If we take into account all the B models involved in this case study (i.e. the interfaces of figure 4 and the refinements of figure 7), there were 108 formulas involved in the proof process, among which 30 formulas had to be proved interactively.

5 Related work

One of the first approaches of module reuse trought interface adaptation is the approach of Purtilo and Atlee [14]: they use a dedicated language (called Nimble) for relating a required interface to a provided interface, where the adaptation is made by the developer. Our approach is similar up to the formalism used for representing the interfaces: instead of a dedicated language, we use UML and the B method. Our approach thus has 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 our interface adaptation.

Dynamic component adaptation [12,7] 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 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 [15], are based on signature algebras and theory category. Our approach is rather practical because we chose 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 nonetheless 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 [8]. 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] Abrial, J.-R., "The B Book," Cambridge University Press, 1996.

[2] Behm, P., P. Benoit and J. Meynadier, *METEOR: A Successful Application of B in a Large Project*, in: *Integrated Formal Methods, IFM99*, LNCS **1708** (1999), pp. 369–387.

[3] Bert, D., 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*, LNCS **2805** (2003), pp. 94–113.

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

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

[6] Hatebur, D., M. Heisel and J. Souquières, *A Method for Component-Based Software and System Development*, in: I. C. Society, editor, *Proceedings of the 32tnd Euromicro Conference on Software Engineering And Advanced Applications*, 2006.

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

[8] Lanoix, A., D. Hatebur, M. Heisel and J. Souquières, *Enhancing Dependability of Component-based Systems*, Technical report, LORIA (2006).

[9] Ledang, H. and J. Souquières, *Modeling class operations in B: application to UML behavioral diagrams*, in: *ASE'2001 : 16th IEEE International Conference on Automated Software Engineering* (2001).

[10] Meyer, E. and J. Souquières, *A systematic approach to transform OMT diagrams to a B specification*, in: *Proceedings of the Formal Method Conference*, LNCS 1708 (1999), pp. 875–895.

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

[12] Mtzel, K. and P. Schnorf, *Dynamic component adaptation* (1997).

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

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

[15] Smith, D. R., *Constructing specification morphisms*, Journal of Symbolic Computation **15** (1993), pp. 571–606.

[16] Steria – Technologies de l'information, "Obligations de preuve: Manuel de référence, version 3.0," (1998).

A Source code of B models and refinements

```
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
```

Figure A.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
```

Figure A.2. B model of Alarms_U

```
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
```

Figure A.3. Alarms_DB_ref1: first step of the refinement

```
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
```

Figure A.4. Alarms_DB_ref2: second step of the refinement

```
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
```

Figure A.5. Alarms_DB_ref3: third step of the refinement

```
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
```

Figure A.6. Alarms_DB_ref4: fourst step of the refinement

```
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
```

Figure A.7. Alarms_DB_ref5: last step of the refinement