

Electronic Notes in Theoretical Computer Science 203 (2009) 23–35

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

Trustworthy Interface Compliancy: Data Model Adaptation Using B Refinement

Samuel Colin, Arnaud Lanoix, Jeanine Souqui`eres

*LORIA – Universit´e Nancy 2 Campus Scientifique, BP 239*

*F-54506 Vandœuvre l`es Nancy cedex*

[*{*Samuel.Colin,Arnaud.Lanoix,Jeanine.Souquieres*}*@loria.fr](mailto:Samuel.Colin@loria.fr)

**Abstract**

In component-based software development approaches, components are considered as black boxes, commu- nicating 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.

# 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 de- veloping it from scratch [[22](#_bookmark42)]. 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](#_bookmark26),[4](#_bookmark24),[9](#_bookmark27)] 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](#_bookmark0). Each component interface is equipped with a suitable data model defining all the types occurring in the signatures of interface operations.

1571-0661/© 2009 Elsevier B.V. Open access under [CC BY-NC-ND license.](http://creativecommons.org/licenses/by-nc-nd/3.0/)

doi:10.1016/j.entcs.2009.03.024

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](#_bookmark8), 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](#_bookmark10). The paper finishes with the discussion of related work in section [5](#_bookmark20) and concluding remarks in section [6](#_bookmark21).

# 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.

* 1. *The B method*

B is a formal software development method based on set theory, which supports an incremental development process using refinement [[1](#_bookmark22)]. 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, pre- serving 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 com- plex real-life applications, such as the METEOR project [[2](#_bookmark23)]. 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](#_bookmark25)]. 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 Ate- lierB [[21](#_bookmark43)] or B4free [[6](#_bookmark28)]. Checking POs with B support tools is an efficient and practical way to detect errors introduced during development and to validate the B models.

* 1. *Specifying component architectures*

We define component-based systems using UML 2.0 composite structure diagrams [[16](#_bookmark37)]. 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 confi- dence 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 inte- grated development process, the B models can be obtained by applying systematic derivation rules from UML to B [[14](#_bookmark35),[12](#_bookmark33)].

* 1. *Proving interoperability of component interfaces*

The components must be connected in an appropriate way. To guarantee inter- operability 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 in- terface, and consequently that the two components are compliant as intended [[4](#_bookmark24)].

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](#_bookmark36)], 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.

* 1. *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](#_bookmark1) using the syntax of composite structure diagrams. It uses three COTS components:

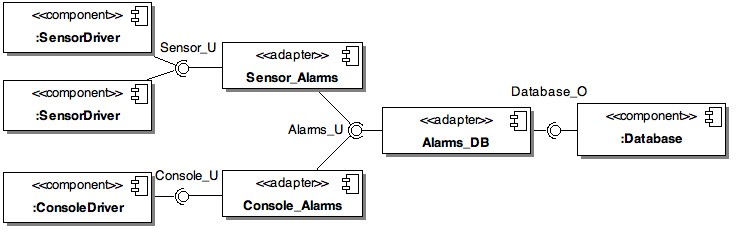


Fig. 1. Component architecture

* Database provides database functionalities described by its provided interface Database O as presented Figure [2](#_bookmark2) 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](#_bookmark3): (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.

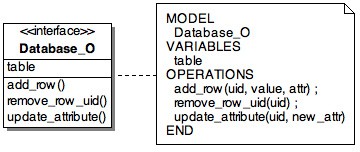


Fig. 2. Interface Database O provided by Database

Listing 1: B model of Database O

**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**;

**END**

* 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.

The interface Alarms U, described in Figure [3](#_bookmark4) and Listing [2](#_bookmark5), expresses the global requirement of the alarms shared between the sensors and the console. Listing [3](#_bookmark6) presents the types used in Alarms U.

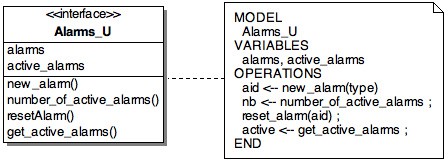


Fig. 3. Interface Alarms U

**MODEL** Types

**SETS**

DeviceIds ; AlarmIds; AlarmTypes;

AlarmStatus = *{*Inactive , Active*}*

**END**

Listing 3: The types used in the development



**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

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](#_bookmark1) and [4](#_bookmark7)).

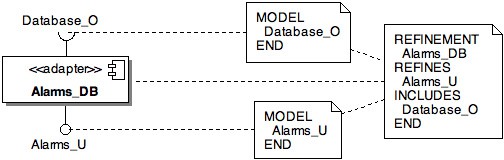


Fig. 4. Adapter Alarms DB

* 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 in a similar way to [[15](#_bookmark36)], as shown Figure [4](#_bookmark7).

# 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 [5](#_bookmark9). 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).

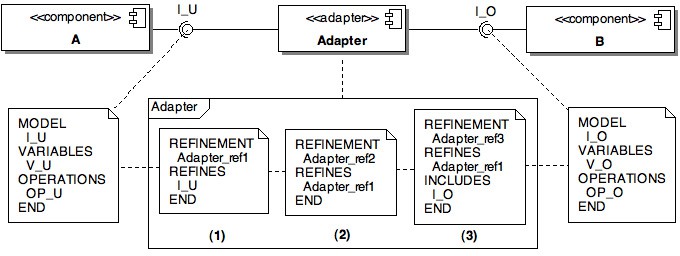


Fig. 5. Process of the adapter development

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.

* 1. *Process description*

The adaptation process is guided by the interface I O and consists of three refine- ment steps. Each step is proved by using the B refinement mechanism.

## 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’, “correspond- ing 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’.

## 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”.

## 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.
  1. *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 con- stants, 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 de- veloper 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.

# 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 [6](#_bookmark11). 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](#_bookmark12) .

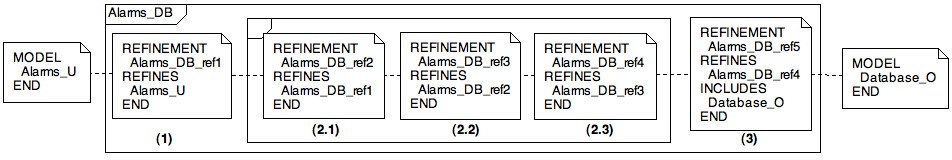


Fig. 6. Refinement steps of the adapter Alarms DB

* 1. *Variables adaptation*

1 Complete B models are published in [[7](#_bookmark29)].

The first step consists in adapt- ing the variables alarms and ac- tive alarms of the interface data model of Alarms U to the inter- face data model of Database O. During this step, we do not in- troduce 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

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

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](#_bookmark13). The proof of this refinement consists of 18 POs, among which 4 have been proved interactively.

* 1. *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).

**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**

* + 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

Listing 5: Step **(2.1)** of the adap- tation process

a new variable nat ids and we add a relation- ship between both variables in the invariant. The other variables are unchanged, and the result is shown in Listing [5](#_bookmark14). 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.

* + 1. *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 pre- sented in Listing [6](#_bookmark15). The invariant helps relat- ing 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 re- finement consists of 14 POs, among which 5

**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**

have been proved interactively.

* + 1. *Typecasting the codomain of each functional variable*

Before this step, the codomains of nat status

Listing 6: Step **(2.2)** of the adap- tation process

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.

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 nat- ural numbers, the codomains of nat status and nat type are trans- formed by the typecasting func- tions mentioned above. For nota- tion consistency, we rename nat ids into ids nn. Moreover, we intro- duce a new variable uid gen for pro- ducing a new unique index each

**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**

time a new alarm is added in the database. All these transforma-

Listing 7: Step **(2.3)** of the adaptation pro- cess

tions are shown in Listing [7](#_bookmark16). 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 re-

placed by all the constants and vari- ables we introduced along the refine- ments.

We have: alarm status = (status cast*−*1

* status nn *◦* id cast). The functions (status nn *◦*id cast) and (status cast*◦*

alarms ids

AlarmStatus

**(1)**

alarms status

id cast

**(2.1)**

**(2.3)**

status cast

nat ids

1*..*card(AlarmStatus)

status nn

**(2.3)**

alarm status) commute. This property is illustrated by Figure [7](#_bookmark17).

Fig. 7: Commutation diagram

* 1. *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, At- tribute and Value of table as il- lustrated in Listing [8](#_bookmark18). We also perform the operation calls to Database O to express the op- erations of Alarms U: the body of the operation new alarm con- sists mainly of a call to the op- eration add row of Database O. The proof of this refinement consists of 19 POs, among which 5 have been proved inter- actively.



**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 **OPERATIONS**

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

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.

|  |  |  |  |
| --- | --- | --- | --- |
|  | *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** |

The proof process for the development of this example, including the proofs of the consistency of the B models of the interfaces (Listings [1](#_bookmark3) and [2](#_bookmark5)) and the proofs of the differ-

ent refinement steps (Listings

Table 1: Proof summary

[4](#_bookmark13), [5](#_bookmark14), [6](#_bookmark15), [7](#_bookmark16) and [8](#_bookmark18) ), is composed

of 108 POs, among which 30 POs have been proved inter- actively (see Table [1](#_bookmark19) for details).

# Related work

One of the first approaches of module reuse through interface adaptation is the approach of Purtilo and Atlee [[17](#_bookmark38)]: 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](#_bookmark34),[10](#_bookmark31)] goes further than our approach by propos- ing methods for adapting *at run-time* components by finding suitable adapter com- ponents based on the interfaces of the components to adapt. Unfortunately these methods have strong requirements (knowing inheritance relationships, runtime map- ping 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](#_bookmark30)] 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 refine- ment between the required interface and the provided one.

Zaremski and Wing [[23](#_bookmark44)] propose an interesting approach to compare two soft- ware 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](#_bookmark39),[19](#_bookmark40)] present adapters in the context of concurrent systems. They consider only a certain class of protocol interoperability problems and gen- erate 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](#_bookmark41)], 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.

# 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 verifi- cation 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](#_bookmark32)]. 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 nor- mal 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 architec- ture 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. M. 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. Boulm´e, 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. Souqui`eres, *Proving component interoperability with B refinement*, Electronic Notes in Theoretical Computer Science (ENTCS) **160** (2006), pp. 157–172.
5. Chouali, S. and J. Souqui`eres, *Verifying the compatibility of component interfaces using the B formal method*, in: *International Conference on Software Engineering Research and Practice (SERP’05)* (2005), pp. 850–856.
6. Clearsy, *B4free*, website (2004), [http://www.b4free.com](http://www.b4free.com/).
7. Colin, S., A. Lanoix and J. Souqui`eres, *Trustworthy Interface Compliancy: Data Model Adaptation*, Research Report hal-00123884, LORIA (2007), <http://hal.archives-ouvertes.fr/hal-00123884>.
8. Ehrig, H., 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*, ENTCS **108**, 2004, pp. 53–67.
9. Hatebur, D., M. Heisel and J. Souqui`eres, *A method for component-based software and system development*, in: *Proceedings of the 32nd Euromicro Conference on Software Engineering And Advanced* *Applications* (2006), pp. 72–80.
10. Kniesel, G., *Type-safe delegation for run-time component adaptation*, Lecture Notes in Computer Science **1628** (1999), pp. 351–366.
11. Lanoix, A., D. Hatebur, M. Heisel and J. Souqui`eres, *Enhancing dependability of component-based systems*, in: S. Verlag, editor, *Reliable Software Technologies Ada-Europe 2007*, number 4498 in LNCS (2007), pp. 41–54.
12. Ledang, H. and J. Souqui`eres, *Modeling class operations in B: application to UML behavioral diagrams*, in: *ASE’2001: 16th IEEE International Conference on Automated Software Engineering* (2001), pp. 289–296.
13. M¨atzel, K.-U. and P. Schnorf, *Dynamic component adaptation*, Technical report, Ubilab laboratory, Union Bank of Switzerland, Zu¨rich, Switzerland (1997).
14. Meyer, E. and J. Souqui`eres, *A systematic approach to transform OMT diagrams to a B specification*, in: *Proceedings of the Formal Method Conference*, number 1708 in LNCS (1999), pp. 875–895.
15. Mouakher, I., A. Lanoix and J. Souqui`eres, *Component adaptation: Specification and verification*, in: *Proc. of the 11th Int. Workshop on Component Oriented Programming (WCOP’06), satellite workshop* *of ECOOP*, 2006, pp. 23–30.
16. Object Management Group (OMG), “UML Superstructure Specification,” (2005), version 2.0.
17. Purtilo, J. M. and J. M. Atlee, *Module reuse by interface adaptation*, Software - Practice and Experience

**21** (1991), pp. 539–556.

1. Reussner, R. H. and H. W. Schmidt, *Using parameterised contracts to predict properties of component based software architectures*, in: I. Crnkovic, S. Larsson and J. 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.
2. Reussner, R. H., 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*, Springer-Verlag, Berlin, Germany, 2003 pp. 287–325.
3. Smith, D. R., *Constructing specification morphisms*, Journal of Symbolic Computation **15** (1993),

pp. 571–606.

1. Steria – Technologies de l’information, “Obligations de preuve: Manuel de r´ef´erence, version 3.0,” (1998).
2. Szyperski, C., “Component Software,” ACM Press, Addison-Wesley, 1999.
3. Zaremski, A. M. and J. M. Wing, *Specification matching of software components*, ACM Transaction on Software Engeniering Methodolology **6** (1997), pp. 333–369.