Proving Component Interoperability with B Refinement

Samir Chouali[1], Maritta Heisel[2], Jeanine Souquières[1]

[1] *LORIA - University Nancy 2*
Campus Scientifique
BP 239, F 54506 Vandoeuvre-lès-Nancy Cedex
{Samir.Chouali, Jeanine.Souquieres}@loria.fr

[2] *Universität Duisburg-Essen*
Fachbereich Ingenieurwissenschaften
Institut für Medientechnik und Software Engineering
D-47048 Duisburg
maritta.heisel@uni-duisburg-essen.de

Abstract

We use the formal method B for specifying interfaces of software components. Each component interface is equipped with a suitable data model defining all types occuring in the signature of interface operations. Moreover, pre- and postconditions have to be given for all interface operations. The interoperability between two components is proved by using a refinement relation between an adaptation of the interface specifications.

Key words: Component interoperability, B method.

1 Introduction

In recent years, the paradigm of component orientation [9,19] has become more and more important in software engineering. Its underlying idea is to develop software systems not from scratch but by assembling pre-fabricated parts, as is common in other engineering disciplines. Component orientation has emerged from object orientation, but the units of deployment are usually more complex than simple objects. As in object orientation, components are encapsulated, and their services are accessible only via interfaces and their operations.

In order to really exploit the idea of component orientation, it must be possible to acquire components developed by third parties and assemble them in such a way that the desired behavior of the software system to be implemented is achieved. This approach leads to the following requirements:

(i) The *description* (i.e., specification) of a component must contain sufficient information to decide whether or not to acquire it for integration in a new software system. First, this requirement concerns the access to the component's source code that may not be granted in order to protect the component producer's interests. Moreover, component consumers should not be obliged to read the source code of a component to decide if it is useful for their purposes or not. Hence, the source code should not be considered to belong to the component specification. Second, it does not suffice to describe the interfaces *offered* by a component (called *provided interfaces* in the following). Often, components need other components to provide their full functionality. Hence, also the *required* interfaces must be part of a component specification.

(ii) For different components to interoperate, they must agree on the format of the data to be exchanged between them. Hence, each interface of a component must be equipped with a data model that describes the format of the data accepted and produced by the component. It does not suffice to give only the signature of interface operations (e.g., operation *foo* takes two integers and yields an integer as its result) as is common in current interface description languages. It is also necessary to describe what effect an interface operation has (e.g., operation *foo* takes two integers and yields their sum as a result).

In order to fulfill the above requirements, a component interface specification must contain the following information:

- a data model associated with each required and provided interface of a component (*interface data model*),
- pre- and postconditions for each interface operation, such that design by contract [13] becomes possible.

We use UML class diagrams [4] to express the interface data model and the formal notation B [1]. Based on these ingredients, we prove the interoperability between two components by using a refinement relation between an adaptation of their interface specifications. Part of this notion of interoperability between component interfaces is based on a specification matching approach [24].

We chose to use the B method because its underlying concepts of machine and refinement fit well with components and their interoperability, and because the method is equipped with powerful tool support. Thus, we can exploit existing technology for proving component interoperability. Using for example the object constraint language OCL and generating verification conditions from scratch would be much more tedious.

Note that our approach takes into account only the functional aspects of components. Non-functional aspects such as security and performance are of course also important, and we aim to treat these issues in future work.

The rest of the paper is organized as follows: in Section 2, we discuss related work. Then, we present an overview of the B method in Section 3. We introduce the specification of component interfaces in Section 4. The notion of interoperability between two components is defined in Section 5 with its verification using the notion of refinement as it is defined for B. The case study of a hotel reservation system serves to illustrate our approach. The paper finishes with some concluding remarks in Section 6.

2 Related Work

In an earlier paper, we have investigated the role of component models in component specification [10]. The specification of a component model makes it possible to obtain more concise specifications of individual components, because these may refer to the specification of the component model. The component model specification need not be repeated for each individual component adhering to the component model in question. In this paper, we investigate the necessary ingredients a component specification must have in order to be useful for assembly of a software system out of components. These ingredients are independent of concrete component models. Several proposals for component specification have already been made. They have in common that they have no counterpart of our interface data model and that they do not consider interoperability issues, but only the specification of single components.

A working group of the German "Gesellschaft für Informatik" (GI) has defined a specification structure for business components [20]. That structure comprises seven levels, namely marketing, task, terminology, quality, coordination, behavioral, and interface. Our specification structure covers the layers terminology, coordination, behavioral, and interface by proposing concrete ways of specifying each of those levels. The other layers of the GI proposal have to do with non-functional aspects of components.

Beugnard et al. [3] propose to define contracts for components. They distinguish four levels of contracts: syntactic, behavioral, synchronization, and quality of service. The syntactic level specifies only the operation signatures, the behavioral level contains pre- and postconditions, the synchronization level corresponds to usage protocols, and the quality of service level deals with non-functional aspects. Beugnard et al. do not introduce data models for their interfaces. It cannot easily be checked if two components can be combined.

The component specification approach of Lau and Ornaghi [11] is closer to ours, because there, each component has a *context* that corresponds to our interface data model. A context is an algebraic specification, consisting of a signature, axioms, and constraints. In contrast, we deem it more appropriate to allow for an object-oriented specification of the data model of a component interface. This makes it possible to take side effects of operations into account and to use inheritance, concepts that are frequently used in practice.

Cheesman and Daniels [6] propose a process to specify component-based software. This process starts with an informal requirements description and produces an architecture showing the components to be developed or reused, their interfaces and their dependencies. For each interface operation, a specification is developed, consisting of a precondition, a postcondition and possibly an invariant. This approach follows the principle of design by contract [13]. Our specification of component interfaces is inspired by Cheesman and Daniels' work because that work clearly shows that for each interface, a data model is necessary. However, Cheesman and Daniels do not consider the case that already existing components with possibly different data models have to be combined, and hence they do not define a notion of interoperability.

Canal et al. [5] use a subset of the polyadic π-calculus to deal with component interoperability only at the protocol level. The π-calculus is well suited for describing component interactions. The limitation of this approach is the low-level description of the used language and its minimalistic semantics.

Bastide et al. [2] use Petri nets to specify the behavior of CORBA objects, including operation semantics and protocols. The difference with our approach is that we take into account the invariants of the interface specifications.

Zaremski and Wing [24] 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.

Others [8,21] have also proposed to enrich component interface specifications by providing information at signature, semantic and protocol levels. Despite these enhancements, we believe that in addition, a data model is necessary to perform a formal verification of interface compatibility.

The idea to define component interfaces using B has been introduced in an earlier paper [7].

3 The B method

The B method [1] is a formal software development approach allowing to develop software for critical systems. It covers the entire development process from an abstract specification to an implementation. Its basis is set theory. The basic building block is the *abstract machine* that is similar to a module or a class in an object-oriented development. A B specification consists of one or several abstract machines (examples of B machines are given in Section 4). Each of them describes a set of variables, invariance properties (also called safety properties) referring to these variables, an initialization, which is a predicate initializing the variables, and a list of operations. The specification of an operation consists of a precondition part and a body part. The precondition expresses the requirement that must be met whenever the operation is called. The body expresses the effect of the operation. The states of a specified system are only modifiable by operations that must preserve its invariant. A B operation OP is defined as : $OP \stackrel{\text{def}}{=} \text{PRE } P \text{ THEN } S \text{ END}$, where P is a precondition, and S is the body part, expressed as a generalized substitution. S may for example take the following shapes:

- assignment statement: $S \stackrel{\text{def}}{=} x := E$ where x is a variable and E is an expression,
- multiple assignment: $S \stackrel{\text{def}}{=} x, \ldots, y := E, \ldots, F$,
- IF statement: $S \stackrel{\text{def}}{=} \text{IF } P' \text{ THEN } S' \text{ ELSE } T' \text{ END}$, where P' is a predicate, S' and T' are substitutions.

The formula $[S]Post$ (where S is a substitution, and $Post$ is a predicate) is called the weakest precondition for S to achieve $Post$. It denotes the predicate which is true for any initial state, from which the execution of S is guaranteed to achieve $Post$.

The B method provides structuring primitives that allow one to compose machines in various ways. Large systems can be specified in a modular way and in an object-based manner [14,12]. A system is developed by refinement used to transform an abstract specification step by step into more concrete ones. For each refinement step, we have to prove that the refined specification is correct with respect to the more abstract specification. In the end, we arrive at an implementation that refines its abstract specification. Verification can be done with the B theorem prover, Atelier B [18].

4 Specification of component interfaces

Our goal is to propose a way of specifying components as black boxes, so that component consumers can deploy them without knowing their internal details. Hence, component interface specifications play an important role, as interfaces are the only access points to a component.

4.1 Definition

A component specification must contain all information necessary to decide whether the component can be used in a given context or not. This concerns the data used by the component as well as its behavior visible to its environment. This behavior is realized by services which can be used by other components or software systems. These services are collected in *provided interfaces*. However, in many cases, a component depends on services offered by other components. In this case, the component can work correctly only in the presence of other components offering the required services. The services required by a component are collected in *required interfaces*. Required interfaces are an important part of a component specification, because without the knowledge what other components must be acquired in addition, it is impossible to use the component in a component-based system. An interface specification consists of the following parts:

(1) The specification of its interface data model which specifies (i) the types used in the interface, (ii) a data state as far as necessary to express the effects of operations, and (iii) invariants on that data state. In the following, we use UML class diagrams [4] to express the data model. This class diagram is then automatically transformed into a B specification [14]. Other languages, such as Object-Z [17], are also suitable for specifying the interface data model (see [10]).

(2) A set of operation specifications. An operation specification consists of its signature (i.e., the types of its input and output parameters), its precondition expressing under which circumstances the operation may be invoked, and its postcondition expressing the effect of the operation. Both pre- and postcondition will refer to the interface data model.

For each component interface, a B machine is defined that contains speci-

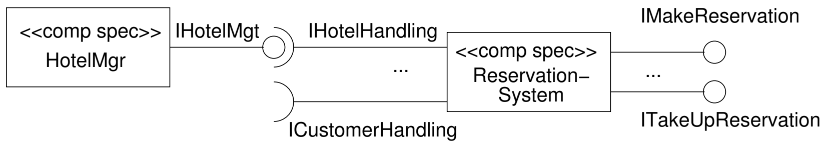

Fig. 1. Component architecture of the hotel reservation system

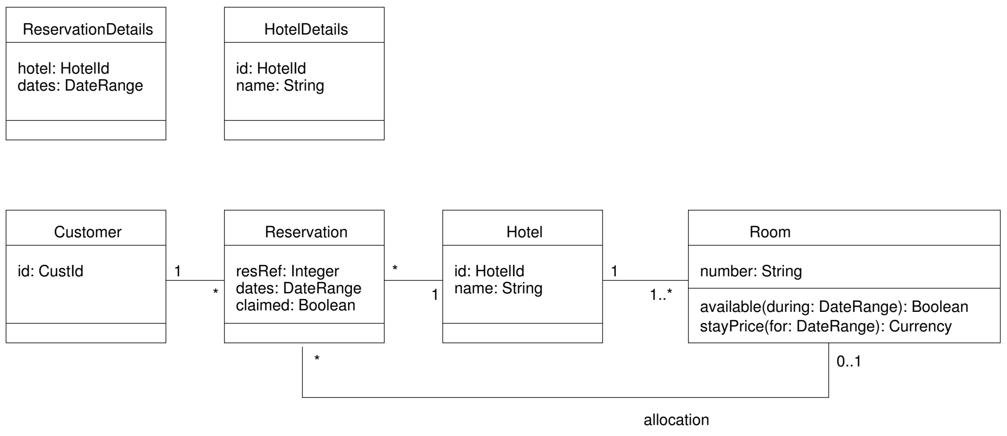

Fig. 2. Interface data model of *IHotelHandling*

fications of the interface data model and of the operations.

4.2 Case study

We illustrate our approach by considering a hotel reservation system, a variant of the case study used by Cheesman and Daniels [6]. The architecture of the global reservation system using components is described in Fig.1 using UML 2 notation [15]. It has two export interfaces, *IMakeReservation* and *ITakeReservation*, and two import interfaces *IHotelHandling* and *ICustomerHandling*. One of the used components is *HotelMgr* with its export interface *IHotelMgt*.

In the following, we will consider the interfaces *IHotelHandling* and *IHotelMgt* in more detail in order to prove that the component *HotelMgr* with its interface *IHotelMgt* satisfies the needs of the interface *IHotelHandling*.

4.2.1 Specification of the interface IHotelHandling

Figure 2 shows the interface data model, expressed as a class diagram.

The corresponding B specification is obtained by systematic transformation rules applied on the UML class diagram in the following way. Since in B all variables must have different names, we use the naming convention that all variable names are prefixed by an abbreviation of the name of the class they belong to. For example, the attribute *hotel* of the class *ReservationDetails* becomes the variable *RD_hotel* in the B machine *IHotelHandling*.

Classes. As we can see in Fig. 3, the classes of the interface data model and the types of their attributes are represented as sets. Attributes are defined as variables which are functions. The sets of objects that exist in the system, such $cust, res, hotels$ and $rooms$ are also defined as variables. For example, $cust$ is declared to be a subset of the set $Customer$.

Associations between classes. They are specified as variables whose type is a function or relation (depending on the multiplicities of the association) between the sets that model the associated classes. Figure 4 shows the B specification of the associations between the classes: $Reservation$ and $Customer$, $Reservation$ and $Hotel$, $Reservation$ and $Room$.

Integrity constraints. They are specified as predicates in the INVARIANT clause of the B machine. For example, the constraint which expresses that a reservation is claimed if and only if a room is allocated to it is expressed as: $\forall(re).((re \in res) \Rightarrow ((RES\_claimed(re) = TRUE) \Leftrightarrow (re \in dom(assoc\_Allocation))))$

MACHINE $IHotelHandling$

SETS $ReservationDetails; HotelId; DateRange; HotelDetails;$

$Customer; CustID; Reservation; Hotel; Room; Currency$

VARIABLES $RD\_hotel, RD\_dates, HD\_id, HD\_name, C\_id, cust, RES\_resRef, RES\_dates,$

$RES\_claimed, RES\_number, Hotel\_id, Hotel\_name, hotels, res, R\_number,$

$R\_available, R\_stayPrice, rooms$

INVARIANT

$/ * classReservationDetails * /$

$RD\_hotel \in ReservationDetails \rightarrow HotelID \ \wedge \ RD\_dates \in ReservationDetails \rightarrow DateRange \ \wedge$

$/ * classHotelDetails * /$

$HD\_id \in HotelDetails \rightarrow HotelID \ \wedge \ HD\_name \in HotelDetails \rightarrow STRING \ \wedge$

$/ * classReservation * /$

$RES\_resRef \in Reservation \rightarrow INTEGER \ \wedge \ RES\_dates \in Reservation \rightarrow DateRange \ \wedge$

$RES\_claimed \in Reservation \rightarrow BOOL \ \wedge \ RES\_number \in INTEGER \ \wedge$

$/ * classHotel * /$

$Hotel\_id \in Hotel \rightarrow HotelId \ \wedge \ Hotel\_name \in Hotel \rightarrow STRING$

$/ * state \ \ of \ \ the \ \ system * /$

$cust <: Customer \ \wedge \ hotels <: Hotel \ \wedge \ res <: Reservation \ \wedge \ rooms <: Room \ \ldots$

Fig. 3. B specification of the classes in $IHotelHandling$

VARIABLES $\ldots$

$assoc\_ResCust, assoc\_ResHot, assoc\_Allocation$

INVARIANT $\ldots$

$assoc\_ResCust \in Reservation \rightarrow Customer \ \wedge \ assoc\_ResHot \in Reservation \rightarrow Hotel \ \wedge$

$assoc\_Allocation \in Reservation \nrightarrow Room$

Fig. 4. B specification of associations between classes

Class operations. Operations $R\_availabale$ and $R\_stayPrice$ of the class

Room are specified as variables whose type is a function as expressed in the INVARIANT clause of the B machine as follows:
$R\_available \in Room \times DateRange \rightarrow BOOL$
$R\_stayPrice \in Room \times DateRange \rightarrow Currency$

Operations. They are specified in the *OPERATIONS* clause of the B machine. Figure 5 gives two examples of operations: *getHotelDetails* yields at its result a collection of hotel details, where the hotel name must match the input parameter *match*; *makeReservation* creates a reservation, given some customer and some reservation details. It has the precondition that the hotel contained in the reservation details actually exists. The notation "||" denotes a parallel assignment.

```
OPERATIONS ...
hotdets ← getHotelDetails(match) ≙
PRE match ∈ STRING
THEN hotdets := {hdx|hdx ∈ HotelDetails ∧ ∃ho.((ho ∈ Hotel) ∧ (HD_id(ho) = HD_id(hdx)) ∧
        (HD_name(ho) = HD_name(hdx)) ∧ (matches(match, HD_name(hdx)) = TRUE))}
END;
resref ← makeReservation(pres, cus) ≙
PRE pres ∈ ReservationDetails ∧ cus ∈ CustID ∧ ∃ho.(ho ∈ hotels ∧ HD_id(ho) = RD_hotel(pres))
THEN ANY ho WHERE (ho ∈ hotels ∧ H_id(ho) = RD_hotel(pres)) THEN
        ANY nres WHERE nres ∈ Reservation ∧ nres ∉ res ∧
        nres ∉ dom(assoc_Allocation) ∧ nes ∉ dom(assoc_ResHot) THEN
        res := res ∪ nres || assoc_ResHot(nres) := ho || C_id(assoc_ResCust(nres)) := cus
        || resref := RES_number + 1 || RES_resRef(nres) := RES_number + 1
        || RES_dates(nres) := RD_dates(pres) || RES_claimed(nres) := FALSE   END END
```

Fig. 5. B specification of operations

All that information is collected in a single abstract B machine, called *IHotelHandling* is available at http://www.loria.fr/˜chouali/specB.

4.2.2 Specification of the interface IHotelMgt

We assume that a component *HotelMgr* is available that can manage hotels with different kinds of rooms. Figure 6 shows the interface data model for its provided interface *IHotelMgt*.

The differences between the interface *IHotelHandling* and the interface *IHotelMgt* are due to the new class *RoomType* in *IHotelMgt* to take into account different kinds of rooms. All the classes present in the interface data model of *IHotelHandling* are also present in the interface data model of *IHotelMgt*. However, the classes *ReservationDetails* and *HotelDetails* now have one more attribute related to *RoomType*.

In the following, we only show a part of the B specification of the interface *IHotelMgt* that expresses the changes as compared to *IHotelHandling*.

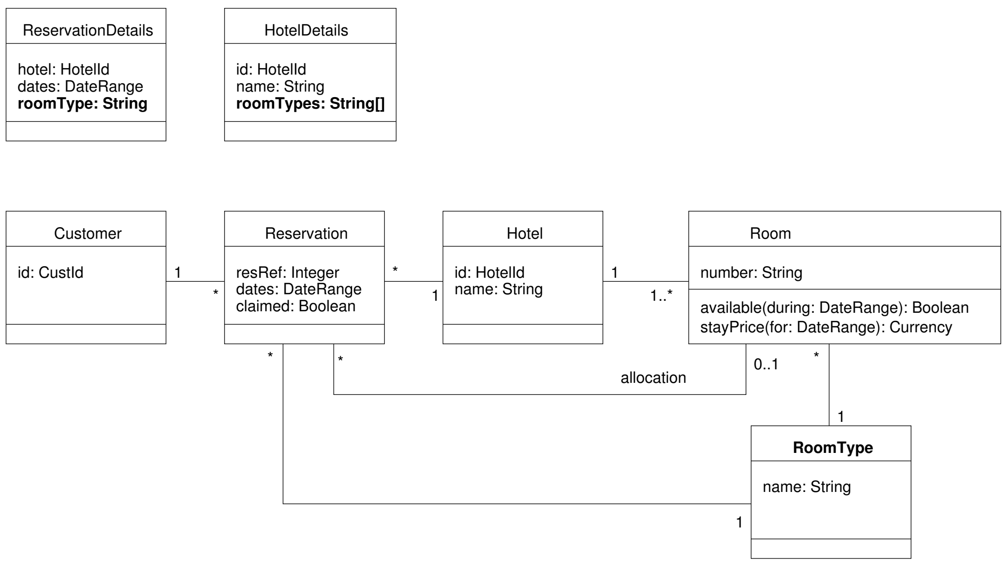

Fig. 6. Interface data model of $IHotelMgt$

The class ***RoomType*** **and its associations.** Figure 7 presents the specification of the class *RoomType*, its attribute and the associations between the classes *Room* and *Roomtype*, *Reservation* and *RoomType*.

MACHINE $IHotelMgt$

SETS ...

$RoomType$

VARIABLES ...

$RT\_name, RD\_roomType, HD\_roomTypes$

INVARIANT ...

$/*\ classRoomType\ */$

$RT\_name \in RoomType \rightarrow STRING \ \wedge \ RD\_roomType \in ReservationDetails \rightarrow STRING \ \wedge$

$HD\_roomTypes \in HotelDetails \rightarrow POW(STRING)$

$/*\ associations\ */$

$assoc\_RoomRt \in Room \rightarrow RoomType \ \wedge \ assoc\_ResRt \in Reservation \rightarrow RoomType \ \wedge \ \ldots$

Fig. 7. B specification of the class *RoomType* in $IHotelMgt$

Invariant properties. The operations must respect two important invariant properties:

- for each object of the class *ReservationDetails* which is associated to an object of the class *Hotel*, the value of its variable *RD_roomType* is the value of the attribute *name* of an object of type *RoomType* associated to a room that belongs to the hotel:
  $\forall (pres, ho).(pres \in ReservationDetails \ \wedge$
  $ho \in hotels \ \wedge \ HD\_id(ho) = RD\_hotel(pres) \ \Rightarrow$
  $RD\_roomType(pres) \in \{rtn | rtn \in STRING \ \wedge$
  $\exists (rty, ro).((rty \in RoomType) \ \wedge \ ro \in Room \ \wedge \ ro \in assoc\_ResHot^{-1}[\{ho\}] \ \wedge$

$assoc\_RoomRt(ro) = rty \ \wedge \ RT\_name(rty) = rtn)\})$

- for each object of class *HotelDetails* which is associated to a hotel, the value of its variable *RD_roomTypes* is the set of the attribute *name* of the objects of class *RoomType* associated to a room that belongs to the hotel:
  $\forall(hdx, ho).(hdx \in HotelDetails \wedge ho \in hotels \wedge \ HD\_id(ho) = HD\_id(hdx) \ \Rightarrow$
  $RD\_roomTypes(hdx) = \{rtn | rtn \in STRING \ \wedge \ \exists(rty, ro).$
  $((rty \in RoomType) \ \wedge \ ro \in Room \ \wedge \ ro \in assoc\_ResHot^{-1}[\{ho\}] \wedge$
  $assoc\_RoomRt(ro) = rty \ \wedge \ RT\_name(rty) = rtn)\})$

Operations. The interface *IHotelMgt* offers operations having the same names as in the interface *IHotelHandling*. However, the specification of the operation *makeReservation* is different from the specification of the same operation in *IHotelHandling*, due to the class *RoomType* (see Fig. 8).

```
OPERATIONS ...
resref ← makeReservation(pres, cus) ≙
PREpres ∈ ReservationDetails ∧ cus ∈ CustID ∧ ∃ho.(ho ∈ hotels ∧ H_id(ho) = RD_hotel(pres))
THEN ANY ho WHERE (ho ∈ hotels ∧ HD_id(ho) = RD_hotel(pres)) THEN
    ANY romt, ro WHERE romt ∈ RoomType ∧ RT_name(romt)= RD_roomType(pres) ∧
    ro ∈ rooms ∧ ro ∈ assoc_RHot ⁻¹[{ho}] ∧ assoc_RoomRt(ro) = romt
    THEN ANY nres WHERE nres ∈ Reservation ...(see figure 5)
    || assoc_ResRt(nres):= romt END END
END
```

Fig. 8. *makeReservation* in *IHotelMgt*

5 Interoperability between Components

Interoperability means the ability of two or more components to communicate and cooperate despite differences in their implementation language, the execution environment, or the model abstraction [22]. Three main levels of interoperability have been distinguished: (i) The signature level (signature of operations); this level covers the static aspects of component interoperation. (ii) The semantic level (meaning of operations); this level covers the behavioral aspects of component interoperation. (iii) The protocol level: this level deals with the order in which a component expects its methods to be called.

In this paper we only deal with the verification of component interoperability at signature and semantic levels. Interoperability at protocol level is treated in [7]. Checking component interoperability is crucial for component-based software development and modification, because it allows system designers and implementors to determine whether two components can interoperate or whether one component can be replaced by another one. Since components are described by their interfaces, verifying component interoperability must be performed on the level of component interfaces. Therefore, in order to verify that two components interoperate, it is necessary to verify that their interfaces are *compatible*.

To fully exploit the advantages of the component-based approach, it must be possible to check the compatibility of two interfaces relying on their specifications only and ignoring implementation details. Our approach to specifying component interfaces given in Section 4 has been designed in such a way that a notion of compatibility can be based on it in a straightforward way.

5.1 Definitions

We first give an intuitive description of component interface compatibility in Sections 5.1.1 and 5.1.2. We then show how that intuitive notion can be mapped to refinement in B (Section 5.1.3).

The provided interface PI of a component C' can play the role of the required interface RI of a component C, if their interface data models and their operations are compatible.

5.1.1 Compatibility of interface data models

The basic idea of compatibility between interface data models is that the interface data model (IDM) of RI is not more restrictive than the one of PI. Only in this case, the IDM of PI can be used in place of the IDM of RI. In particular, each type or class of RI must have a counterpart in PI, but not necessarily vice versa. This means that PI may contain data that are not needed to implement RI. The following cases have to be distinguished:

- Basic types are compatible if they have the same name, or there is an explicit rule stating that the two types are compatible.
- For classes, the following conditions must hold:

(i) For each class $class_r$ of the IDM of RI, there exists a class $class_p$ in the IDM of PI such that
  - For each attribute of $class_r$, there exists an attribute of $class_p$ that has a compatible type.
  - For each operation op_r of $class_r$, there exists an operation op_p of $class_p$, such that for each type of the signature of op_r, there is a compatible type in the signature of op_p. [1]
  - There exists an injective function $tr : class_r \rightarrow class_p$, which transforms an object of $class_r$ into an object of $class_p$. It must be possible to transform RI objects into PI objects in order to use them as input parameters of PI operations. The inverse transformation is necessary to transform the output parameters of PI operations into RI objects.

    For data types of PI that have no counterpart in RI, no transformation function is necessary, because such data can be ignored by RI.

(ii) Each association in the IDM of RI has a counterpart in the IDM of PI, whose cardinality constraints are not stronger than in the IDM of PI.

(iii) The invariant inv_p of the IDM of PI implies the transformed invariant $tr(inv_r)$ of the IDM of RI. This condition ensures that the states permit-

[1] This is a simple version of *signature matching*. Different variants of signature matching in an algebraic context are given by Zaremski and Wing [23]. A discussion of signature matching in the context of components can be found in [16].

ted by the IDM of RI are also permitted by the IDM of PI. However, to show the desired implication, it is necessary that both conditions refer to the same data model. Therefore, the data occurring in the invariant inv_r (which belongs to the IDM of RI) must be replaced by their counterparts in the IDM of PI, as defined by the function tr.

5.1.2 Compatibility of operations

For each operation op_r of interface RI there must exist an operation op_p of interface PI such that:

(i) Their signatures are compatible, i.e., for each type of the signature of op_r, there is a compatible type in the signature of op_p.

(ii) The transformed precondition of op_r, $tr(pre(op_r))$ implies the precondition of op_p. As for the implication relation on the IDM invariants required for compatibility of the IDMs, we must transform the data occurring in the precondition of op_r.

(iii) The postcondition of op_p, $post(op_p)$, implies the transformed postcondition of op_r, $tr(post(op_r))$.

This definition of compatibility of operations corresponds to the notion of plug-in-matching as defined by Zaremski and Wing [24].

5.1.3 Verification of the interface compatibility with the B refinement

In this section, we show that it is possible to use refinement in B to prove that two components are compatible at the signature and semantic levels. We first give the definition of refinement in B [1] and then show how component interface compatibility can be mapped to B refinement.

Let M and N be two B specifications. In the following we give the main conditions that must hold between M and N in order to show that N refines M. M is more abstract than N, but it can also be a refinement of some other specification; we refer to M as the abstract specification.

(i) The state variables of a refinement machine must be different from the state variables of the abstract machine.

(ii) The abstract specification M has an initialization T_m that establishes its invariant I_m. A refinement specification N has an initialization T_n and a coupling invariant J_n. So, if N refines M, then T_n is required to establish J_n in a way which is coherent (non-contradictory) with T_m. Formally, T_m is a refinement of T_n, if and only if $\neg[T_m]\neg J_n$ is true for any state that can be reached from T_n.

(iii) Every operation defined in M must be defined in N, i.e., all abstract operations must be refined.

(iv) If an operation OP_n defined in N refines an operation OP_m in M, then OP_m and OP_n must have the same signature.

(v) Let $OP_m \stackrel{\text{def}}{=}$ PRE P_m THEN S_m END and $OP_n \stackrel{\text{def}}{=}$ PRE P_n THEN S_n END be two operations in M and N, respectively. Let I_m be the invariant defined in M and J_n the coupling invariant defined in N. When the operation OP_n is a refinement of the operation OP_m, then the following conditions hold :

- $I_m \ \wedge \ J_n \ \wedge \ P_m \Rightarrow [S_n]\neg[S_m]\neg J_n$, if the operations have no outputs.
- If out_m and out_n are the outputs of respectively OP_m and OP_n then the following condition must hold:
  $I_m \ \wedge \ J_n \ \wedge \ P_m \Rightarrow [S_n[out_n/out_m]]\neg[S_m]\neg(J_n \ \wedge \ out_m = out_n)$.
- $I_m \ \wedge \ J_n \ \wedge \ P_m \Rightarrow P_n$,

  These conditions express that when the operation OP_m is refined by OP_n, then for any refined execution of S_n on a state in which $I_m \wedge J_n \wedge P_m$ holds, there exists an abstract execution of S_m (S_m and S_n are generalized substitutions). We can conclude that for any OP_m refined by OP_n, the precondition of OP_m implies the precondition of OP_n (because the refinement weakens preconditions), and the states in which the postcondition of OP_n holds are linked with the states in which the postcondition of OP_m holds.

Let us now consider the case where M is a B specification of a required interface RI, and N is a B specification of a provided interface PI. Then the refinement conditions of M with N concerning the initialization[2] and the operations imply the conditions for compatibility between required and provided interfaces, i.e., refinement in B is sufficient for interface compatibility.

However, the refinement condition concerning the disjointness of state variables (condition (i)) cannot be guaranteed to hold (and is not necessary for the compatibility of component interfaces). Hence, in order to use B refinement for proving the compatibility between RI and PI, it is necessary to transform the B specification of PI in order to satisfy the refinement condition 1. That transformation is performed as follows:

- the B specification of PI is transformed into a specification New_PI which is a refinement specification of RI,
- New_PI does not contain the sets already defined in both RI and PI,
- the variables defined in both RI and PI are renamed in New_PI,
- the invariant of New_PI consists of the invariant of PI, where the variable renaming has been applied, and a coupling invariant that relates the newly introduced names to their counterparts in RI.

After performing these steps, we can verify that RI is compatible with PI by proving that New_PI refines RI.

5.2 Case study

We want to prove that the required interface *IHotelHandling* is compatible with the provided interface *IHotelMgt* using B refinement. Figure 9 presents a part of the B specification *New_IHotelMgt* obtained by trans-

[2] A reasonable initialization must be chosen when representing component interfaces as B machines, for example, using empty sets.

forming $IHotelMgt$ according to the steps described above. The main changes concern the renaming of the variables that are also defined in $IHotelHandling$, the definition of the coupling invariant and the definition of the sets and invariance properties that are also defined in $IHotelHandling$.

We use the tool Atelier B [18] to verify that $New\_IHotelMgt$ refines $IHotelHandling$. The verification results are as follows.

- Atelier B generated 197 obvious proof obligations and 22 proof obligations for the B specification $IHotelHandling$. All these proof obligations were proven automatically.
- Atelier B generated 243 obvious proof obligations and 13 proof obligations for the B specification $New\_IHotelMgt$. 12 proof obligations were proven automatically, and 1 was easily proven manually.

According to these results, we conclude that $New\_IHotelMgt$ refines $IHotelHandling$. Consequently, the required interface $IHotelHandling$ is compatible with the provided interface $IHotelMgt$ at the signature and semantic levels [3].

REFINEMENT $New\_IHotelMgt$

REFINES $IHotelHandling$

SETS $RoomType$

VARIABLES $RD\_hotelRef, RD\_datesRef, HD\_idRef, HD\_nameRef, C\_idRef, custRef,$
$RES\_resRefRef, RES\_datesRef, RES\_claimedRef, RES\_numberRef, Hotel\_idRef,$
$Hotel\_nameRef, hotelsRef, resRef, R\_numberRef, R\_availableRef, R\_stayPriceRef, RT\_name,$
$RD\_roomType, HD\_roomTypes, assoc\_RoomRt, assoc\_ResRt$

INVARIANT

$/* renaming\ variables */$

$RD\_hotelRef = RD\_hotel \ \wedge\ RD\_datesRef = RD\_dates \ \wedge HD\_idRef = HD\_id \ \wedge$

$HD\_nameRef = HD\_name \ \wedge\ C\_idRef = C\_id \ \wedge\ custRef = cust \ \wedge$

$RES\_resRefRef = RES\_resRef \ \wedge\ RES\_datesRef = RES\_dates \ \wedge$

$RES\_claimedRef = RES\_claimed \ \wedge\ RES\_numberRef = RES\_number \ \wedge$

$Hotel\_idRef = Hotel\_id \ \wedge\ Hotel\_nameRef = Hotel\_name \ \wedge$

$hotelsRef = hotels \ \wedge\ resRef = res \ \wedge\ R\_numberRef = R\_number \ \wedge$

$R\_availableRef = R\_available \ \wedge\ R\_stayPriceRef = R\_stayPrice \ \wedge$

$/* type\ of\ the\ attributes\ related\ to\ RoomType */$

$RT\_name \in RoomType \to STRING \ \wedge\ RD\_roomType \in ReservationDetails \to STRING \ \wedge$

$HD_roomTypes \in HotelDetails \to POW(STRING) \ \wedge\ assoc\_RoomRt \in Room \to RoomType \ \wedge$

$assoc\_ResRt \in Reservation \to RoomType \ \wedge\ \ldots$

Fig. 9. B specification of $New\_IHotelMgt$

[3] To transform objects of the classes $ReservationDetails$ and $HotelDetails$ (function tr), we use a default room type called "Standard".

6 Conclusion and Future Work

We have presented a manner of specifying component interfaces that is independent of specific component models. Based on that specification, we have defined a notion of compatibility between component interfaces that allows one to check whether two components can interoperate via the given interfaces or not. We have shown that it is possible to use refinement in B to prove that two components are compatible at the signature and semantic levels.

In contrast to previous work, our specification contains a data model associated with each component interface. Without such an explicit interface data model, it would not be possible to check the interoperability of components without knowing details of the component's implementation.

To construct a working system out of components, however, it does not suffice just to check our compatibility conditions. Once compatibility is established, the conventions of a chosen component model must be followed in actually combining the components in question. Moreover, glue code, i.e., adapters, have to be developed that implement the transformation of required interface data into provided interface data and vice versa.

In the version of interoperability given in Section 5, the adapters only transform the data and call an operation of the provided interface of the component to be used. However, one can relax the compatibility conditions and use more liberal versions of specification matching, e.g., plug-in-post matching [24]. In this case, an adapter must check if the precondition of the provided operation holds. If not, it has to take appropriate actions other than calling the provided operation. Thus, the construction of adapters becomes a program synthesis problem. This problem becomes more complex for weaker versions of specification matching. In the future, we intend to investigate alternative versions of compatibility and their mappings to refinement in B, and to give patterns for the corresponding adapters.

References

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

[2] R. Bastide, O. Sy, and P.-A. Palanque. Formal specification and prototyping of CORBA systems. In *ECOOP '99, Proc. of the 13th European Conf. on Object-Oriented Programming*, pages 474–494. Springer-Verlag, 1999.

[3] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *IEEE Computer*, pages 38–45, July 1999.

[4] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[5] C. Canal, L. Fuentes, E. Pimentel, J-M. Troya, and A. Vallecillo. Extending CORBA interfaces with protocols. *Comput. J.*, 44(5):448–462, 2001.

[6] J. Cheesman and J. Daniels. *UML Components – A Simple Process for Specifying Component-Based Software.* Addison-Wesley, 2001.

[7] S. Chouali and J. Souquières. Verifying the compatibility of component interfaces using the B formal method. In CSREA Press, editor, *SERP'05, Int. Conf. on Software Engineering Research and Practice*, 2005.

[8] J. Han. A comprehensive interface definition framework for software components. In *The 1998 Asia Pacific software engineering conference*, pages 110–117. IEEE Computer Society, 1998.

[9] G. T. Heineman and W. T. Councill. *Component-Based Software Engineering.* Addison-Wesley, 2001.

[10] M. Heisel, T. Santen, and J. Souquières. Toward a formal model of software components. In C. George and M. Huaikou, editors, *Proc. 4th Int. Conf. on Formal Engineering Methods*, LNCS 2495, pages 57–68. Springer-Verlag, 2002.

[11] K.-K. Lau and M. Ornaghi. A formal approach to software component specification. In G.T. Leavens D. Giannakopoulou and M. Sitaraman, editors, *Proc. of Specification and Verification of Component-based Systems Workshop at OOPSLA2001*, pages 88–96, 2001.

[12] H. Ledang and J. Souquières. Modeling class operations in B: application to UML behavioral diagrams. *ASE'01, 16th IEEE Int. Conf. on Automated Software Engineering*, 2001.

[13] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 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*, number 1708 in LNCS, pages 875—895. Springer-Verlag, 1999.

[15] OMG. UML 2.0 Superstructure Specification. http://www.omg.org, 2004.

[16] R. Rudloff and M. Heisel. Signature matching with UML. 2004.

[17] G. Smith. *The Object-Z Specification Language.* Kluwer Academic Publishers, 1999.

[18] STERIA. *Atelier B : Preuves et Exemples.*

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

[20] K. Turowski, editor. *Standardized Specification of Business Components.* Gesellschaft für Informatik, 2002.

[21] A. Vallacillo, J. Hernandez, and M. Troya. Object interoperability. In *Object Oriented Technology: ECOOP'99 Workshop Reader*, pages 1–21, 1999.

[22] D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst.*, 19(2):292–333, 1997.

[23] A. M. Zaremski and J. M. Wing. Signature matching: a tool for using software libraries. *ACM Trans. on Soft. Engin. and Method.*, 4(2):146–170, 1995.

[24] A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Trans. on Soft. Engin.and Method.*, 6(4):333–369, 1997.