e-Informatica Software Engineering Journal, Volume 2, Issue 1, 2008

Trustworthy Assembly of Components using the B Refinement

Arnaud Lanoix*, Jeanine Souquières*

*LORIA – Nancy-Université, CNRS

Arnaud.Lanoix@loria.fr, 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. In the best cases, the provided interfaces are checked to be compatible with the corresponding required interfaces, but in general, adapters have to be introduced to connect them. We propose to exploit existing notations and languages with their associated tools to specify working systems out of components: UML composite structure diagrams to express the architecture in terms of components and their interfaces, class diagrams, sequence diagrams and protocol state machines to describe the behavior of each component. Component interfaces will then be expressed in B in order to verify the interoperability. The use of B assembling and refinement mechanisms eases the verification of the interoperability between interfaces and the correctness of the component assembly.

1 Introduction

Recent works have shown that assembling components independently produced and taking into account the verification of their assembly with appropriate tools is a promising approach developped since the nineties [14, 21]. The underlying idea is to develop software systems by assembling existing parts [8, 33], as it is common in other engineering disciplines, such as electrical or mechanical engineering. Among the advantages of such approaches, we can cite: (i) reusability of trustworthy software components, (ii) reduction of the development costs due to the reusability, and (iii) flexibility of systems developed by this approach. Assembling components needs to be supported by design methods and verification tools: on one hand, current technologies of components [25, 28, 32, 34] do not take into account safety requirements, on the other hand, development and certification processes of critical software, based on formal methods, is not well suited to component-based approaches.

The development of component-based systems introduces a fundamental evolution in the way systems are acquired, integrated, deployed and modified. Systems can be designed by examining existing components, like COTS or Commercial Off-The-Shelf components, to see how they meet the expected requirements and decide how they can be integrated to provide the expected functionalities. Next, the system is engineered by assembling the selected components with some locally developed pieces of code [10, 20].

Components are seen as black-boxes units which only specify interfaces and explicit dependencies. An interface describes services offered and required by a component without disclosing the component implementation. Component interfaces are the only access to component informations and functionalities. The services offered by a component are described by provided interfaces and the needed services are described by required interfaces.

For different components to be deployed and to work together, they must interoperate: their interfaces must be compatible through different levels of compatibility depending on the requirements of the developed system. The syntactic level covers signature aspects of attributes and methods provided or required by the interfaces whereas the semantic level concerns behavioral aspects of the considered methods and the protocol level covers the allowed sequence of method calls.

The availability of formal languages and tool support for specifying interfaces and checking their compatibility is necessary in order to verify the interoperability of components. Our approach is supported by a rigourous development methodology based on UML and the B method and is introduced at the level of software architecture. The idea to define component interfaces using B has been introduced in an earlier paper [12] : semantics and protocols of the component services can be easily modeled using the B formal method. The use of the B refinement [1] to prove that two components are compatible at the signature and semantics levels has been explored in [11]. To guarantee a trustworthy assembly of components, each connection of a required interface with another provided interface has to be considered. In the best cases, the provided interface constitutes an implementation of the required interface. In general cases, to construct a working system out of components, adapters have to be defined. An adapter is a piece of glue code that expresses the mapping between a required and a provided interface. At the signature level, it must express the mapping between required and provided variables and how the required methods are implemented in terms of the provided ones. In [26], we have proposed a first definition of an adapter in a simple case, with only one required and one provided interface.

In this paper, we generalize the previous results, taking into account a more general assembly of components with the use of both cases of interfaces for different components to be connected. We use the following notations:

- UML 2.0 [27] composite structure diagrams serve to express the overall architecture of the system in terms of components and interfaces.
- UML 2.0 class diagrams serve to express interface data model with its different attributes and operations.
- The usage protocol of each interface can be modeled by a Protocol State Machine (PSM).
- UML 2.0 sequence diagrams serve to express the interactions between the components to be connected.

- The use of the formal method B [1] and its associated tools serve to specify interfaces, giving a special attention to correctness, increasing confidence in the developed systems: correctness of specifications, as well as correctness of the followed process with verification aspects.

In the following, we present the case study of a simple access control system defined in terms of components with a special focus on the identification component, itself defined in terms of components. Section 3 exposes the trustworthy assembly problem in a general manner. Section 3 presents a simple case of trustworthy component assembly. Section 4 presents a more general case of component assembly. Some related works are discussed in Section 5 and Section 6 concludes the paper.

2 Case Study: a Simple Access Control System

We illustrate our purpose with the case study of a simple access control system which manages the access of authorized persons to existing buildings [2]. Persons who are authorized to enter the building have to be identified. The needed authentication informations may be stored on an electronic access card or a sophisticated key or a bar code pass, etc. Turnstiles block the entrance and the exit of each building until an authorization is given whereas identification systems are installed at each entrance and exit of the concerned buildings. The means of identification can be read to kept out the authentication informations. It can be inserted and ejected and must be taken by the user before a fixed time of 30 seconds, else it is retracted and kept by the system.

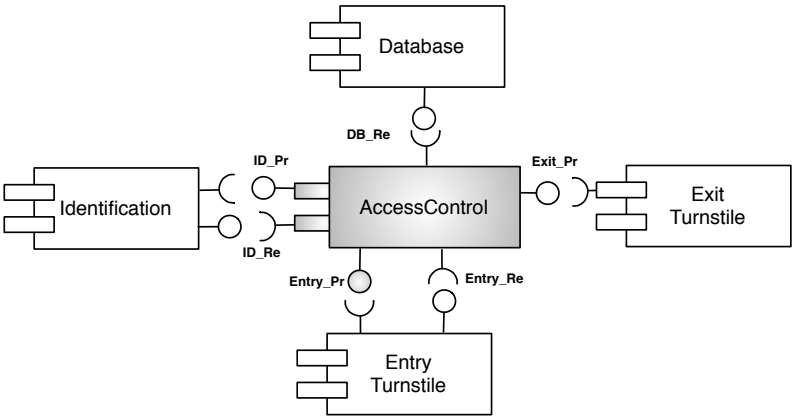
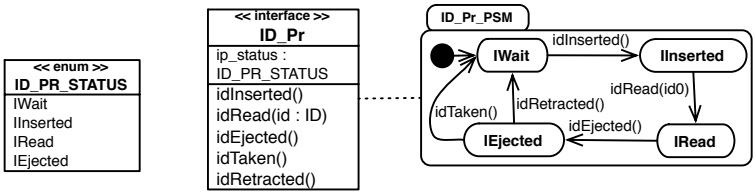


Figure 1: Partial view of the architecture of the access control system

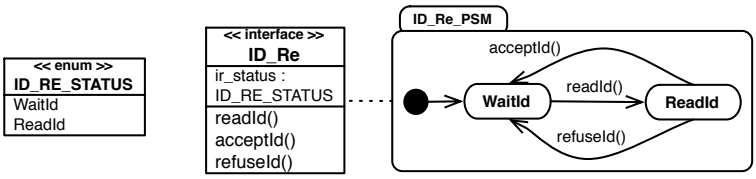
A partial view of the architecture of the access control system is given Figure 1 as a UML 2.0 composite structure diagram. Such diagrams contain named rectangles corresponding to the components of the system; here, we have depicted four components: the AccessControl component corresponding to the system requirements, an Identification component corresponding to the control of the identification, a Database component which is a passive component knowing informations about the authorization of each concerned

user and some Turnstile components. They are connected by means of interfaces which may be required or provided. Required interfaces explicit the context dependencies of a component and are denoted using the "socket" notation whereas provided interfaces explain which functionalities the considered component provides and are denoted using the "lollipop" notation.

We will focus on the interactions between the AccessControl and the Identification components. Requirements concerning the Identification component are expressed by the two interfaces of the AccessControl component. They have been outlined in Figure 2; they are modeled by class diagrams with their different attributes and methods:



(a) Provided interface ID_Pr



(b) Required interface ID_Re

Figure 2: Interfaces of AccessControl related to Identification

- ID_Pr corresponds to its provided interface related to the Identification component with five operations: idInserted, idRead parameterized by some authentication informations represented by id, id Ejected, idTaken and idRetracted.
- ID_Re corresponds to its required interface which must be provided by an Identification component with three operations, readId, acceptedId and refusedId.

Enumerated data types are defined using the stereotype "enum". The usage protocol of each interface is modeled by a Protocol State Machine (PSM) as presented in Figure 2. A PSM specifies the external behavior of the component, with the order of the allowed method calls starting from its initial state.

The B method. It is a formal software development method based on set theory which supports an incremental development process, using refinement [1]. 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 models from textual descriptions, preserving correctness in each step.

The method has been successfully applied in the development of several complex real-life applications, such as the METEOR project [4]. 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 [5]. 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 [22, 24]. Proofs of invariance and refinement are part of each development. The proof obligations are generated automatically by support tools such as AtelierB [31] or B4free [13], an academic version of AtelierB. Checking proof obligations with B support tools is an efficient and practical way to detect errors introduced during development.

```
MODEL
  ID_Pr
SEES
  Type
VARIABLES
  ip_status
INVARIANT
  ip_status ∈ ID_PR_STATUS
INITIALISATION
  ip_status := IWait
OPERATIONS
idInserted =
  PRE ip_status = IWait
  THEN ip_status := IInserted
  END ;
idRead(uid) =
  PRE uid ∈ UID ∧ ip_status = IInserted
  THEN ip_status := IRead
  END ;
idEjected =
  PRE ip_status = IRead
  THEN ip_status := IEjected
  END ;
idTaken =
  PRE ip_status = IEjected
  THEN ip_status := IWait
  END ;
idRetracted =
  PRE ip_status = IEjected
  THEN ip_status := IWait
  END
END
```

(a) Provided interface ID_Pr

```
MODEL
  ID_Re
SEES
  Type
VARIABLES
  ir_status
INVARIANT
  ir_status ∈ ID_RE_STATUS
INITIALISATION
  ir_status := WaitId
OPERATIONS
readId =
  PRE ir_status = WaitId
  THEN ir_status := ReadId
  END ;
acceptId =
  PRE ir_status = ReadId
  THEN ir_status := WaitId
  END ;
refuseId =
  PRE ir_status = ReadId
  THEN ir_status := WaitId
  END
END
```

(b) Required interface ID_Re

Figure 3: B Models for the interfaces of AccessControl

Example. For each interface given in Figure 2, we give a B model as presented in Figure 3: each model consists of a set of variables, invariant properties of those variables and operations. The state of the model, i.e. the set of variables values, is modifiable by operations, which must preserve its invariant:

- in the ID_Pr model, the variable id_status has four possible states and its initial state is IWait. After an idInserted() call, its state is changed to IInserted,
- the ID_Re model expresses the required behaviors of the card reader. A variable, namely ip_status, gives its state, which is initialized with WaitId.

In an integrated development process, the B models can be obtained by applying systematic derivation rules from UML to B [23, 24].

Our purpose is to define the AccessControl component using existing components available on the market.

3 Component Trustworthy Assembly

Components must be assembled in an appropriate way. Interoperability means the ability of components to communicate and to cooperate despite differences in their implementation language, their execution environment, or their model abstraction [35]. Two components are interoperable if all their interfaces are compatible [11]. More precisely, it means that, for each required interface of a considered component, there exists a compatible interface which is provided by another existing component. Three main levels of interfaces compatibility are considered and checked:

- the syntactic level covers static aspects and concerns the interface signature. Each attribute of the required interface must have a counterpart in the provided one; for each method of the required interface, there exists an operation of the provided interface with the same signature,
- the semantic level covers behavioral aspects,
- the protocol level deals with the expression of functional properties (like the order in which a component expects its methods to be called).

A provided interface can propose more functionalities (attributes, methods, behaviors, protocols, etc.) than the required one needs, but all the functionalities used by the required interface must be proposed by the provided one. The process of proving interoperability between components is described in [11].

Often, to construct a working assembly out of components, adapters have to be defined, connecting the required interfaces to the provided ones. An adapter is a new component that realizes the required interface using the provided interface. At the signature level, it expresses the mapping between required and provided variables. At the behavioral and protocol levels, it expresses how the required operations are implemented in terms of the provided ones. In [26], we have studied the adapter specification and its verification using B. We have given a B model of the adaptation that must refine the B model of the required interface including the provided incompatible interface.

More generally, the component assembly concerns the use of both types of interfaces for different components to be connected. We show that the component assembly is a generalization of the adaptation problem: a new specific component is introduced to manage

the needed components. It realizes all the required interfaces of the considered components using their provided interfaces.

Example. Let us use a component Identification1 whose description is given in Figure 4 to answer the requirements of AccessControl presented Section 2. It is a card reader equipped with two lights, a green one and a red one. These lights indicate if the authorization has been accepted (the green light turns on) or denied (the red light turns on). The two lights cannot be turned on at the same time.

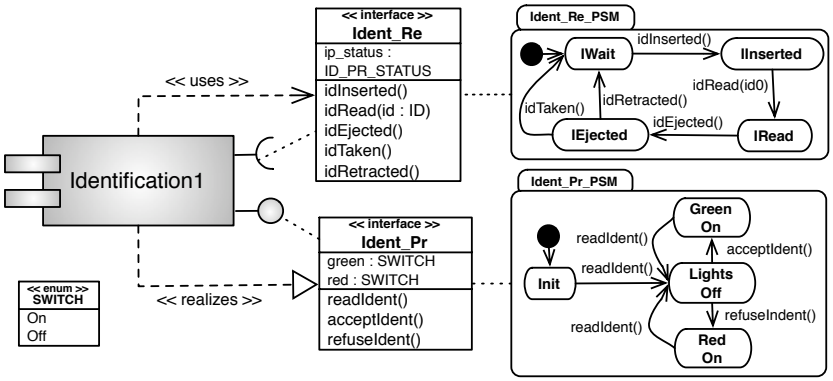


Figure 4: An existing component Identification1

The component Identification1 is equipped with two interfaces:

- Its provided interface Ident_Pr, related to a system controller. Its two variables green and red give the state of two lights: the green light must be turned on if the authorization has been accepted (acceptIdent), otherwise the red light must be turned on (refuseIdent). An invariance property expresses that the two lights cannot be turned on at the same time, as expressed in the B model given in Figure 7(a).
- Its required interface Ident_Re, related to a system controller which is similar to the ID_Pr provided interface of the AccessControl component previously defined.

To ensure that the assembly of components Identification1 and AccessControl is trustworthy [26], we must prove that the corresponding interfaces are compatible, as seen in Figure 5. We decompose this proof into two steps.

3.1 Compatibility between Ident_Re and ID_Pr

We have to prove that ID_Pr *realizes* Ident_Re. This property can be expressed by the B refinement concept. We show that the B model of ID_Pr is a correct refinement of the B model of Ident_Re. That means that the methods of the provided interface implement directly the methods of the required interface. In this example, the proof of this refinement is obvious.

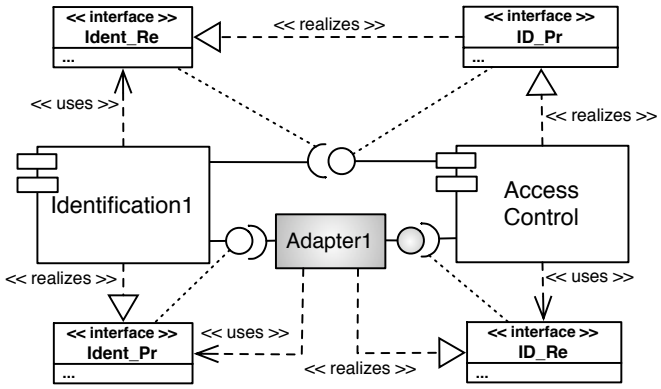


Figure 5: Identification1 and AccessControl assembly

We conclude that the required interface Ident_Re of the component Identification1 is compatible with the provided interface ID_Pr of the AccessControl component. The interoperability is verified at the signature, semantic and protocol levels.

3.2 Adaptation between ID_Re and Ident_Pr

When looking at the required interface ID_Re of the component AccessControl, as expressed in Figure 2, it is obvious that it is not directly compatible with the interface Ident_Pr of the component Identification1, as shown in Figure 4. We propose to introduce a new component, called Adapter1, to map correctly these two incompatible interfaces. In terms of UML, this new component *realizes* ID_Re, *using* Ident_Pr.

The correctness of this adaptation can be proved using the B method. It is expressed by the schema presented Figure 6, in which Adapter1 is modeled by a refinement which:

- *refines* the B model of the required interface ID_Re and
- *includes* the B model of the provided interface Ident_Pr.

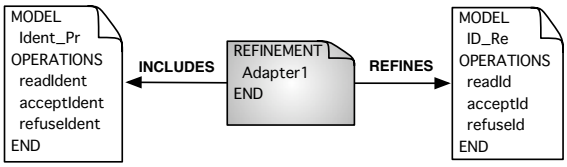
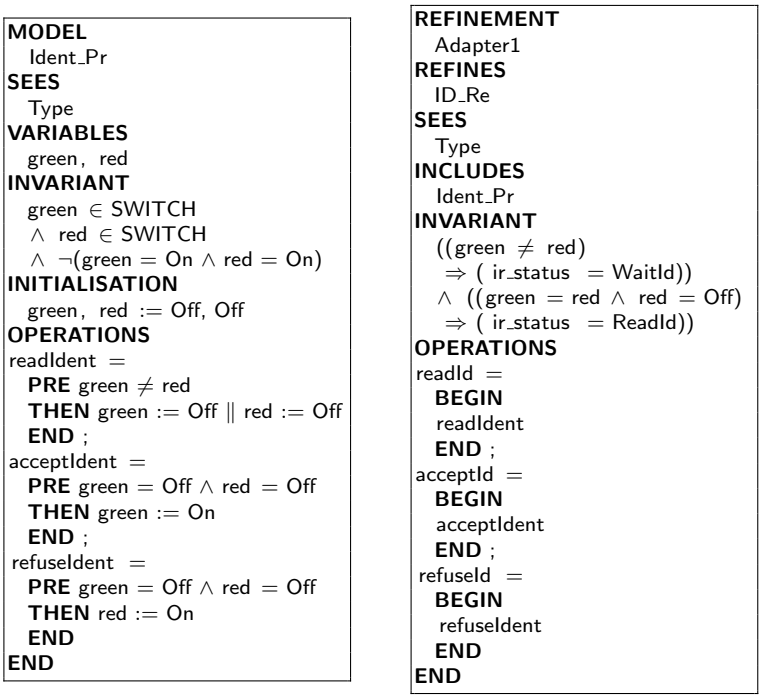


Figure 6: B adaptation between ID_Re and Ident_Pr

The B model of the component Adapter1 proposed in Figure 7(b) expresses the mapping between the two interfaces. The invariant clause, or gluing invariant, makes the correspondence between the required and the provided attributes. The variable ir_status required by the access control is defined in terms of the two variables green and red provided by the identification. The operation clause defines how the required methods, i.e.



(a) B Model of Ident_Pr

(b) B Model of Adapter1

Figure 7: B models for the adaptation

readId, acceptId and refuseId are implemented by the provided ones, readIdent, acceptIdent and refuseIdent.

To prove this refinement, B4free generates 12 obvious proof obligations and 5 proof obligations. As an example, we show in Figure 8 one of these proof obligations concerning the refinement of the method acceptId: we have to prove that red = Off (expressed by red$1 = SW_Off on B4free), using the listed hypotheses. This proof concerns the preservation of the invariant of Adapter1 by the precondition of the used method acceptIndent of Indent_Pr. The 5 proof olbigations are automatically discharged by B4free. As a consequence, Adapter1 implements ID_Re in terms of Ident_Pr. We are able to assemble Identification1 to AccessControl through Adapter1.

4 General Case of Component Trustworthy Assembly

Let us define an identification component corresponding to the previous requirements given section 3 in terms of three existing components, namely CardReader, Timer and MultiLights. The component CardReader is used to read the authentication informations on an access card and the component Timer to indicate the time limit. Green and red lights are provided by the component MultiLights.

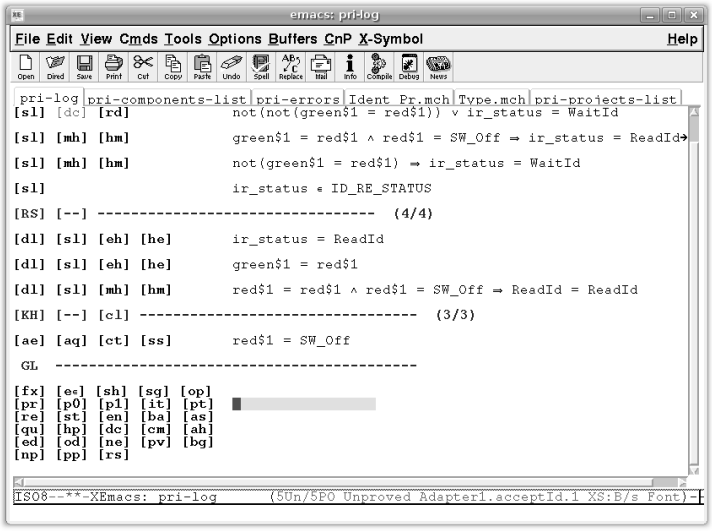


Figure 8: One of the proof obligations of the refinement of acceptId

The general case of component assembly concerns the use of both type of interfaces for different components to be connected. A new specific component is defined to manage these components. It realizes all the required interfaces of the considered components using their provided interfaces.

4.1 Existing components

The functionality of each component is known by its interface descriptions presented below as UML 2.0 diagrams associated to B models for behavioral and protocol specifications.

The component CardReader**.** This component reads identification informations from an access card. It is equipped with two interfaces, as presented in Figure 9, a provided one named Card_Pr with three methods (readCard(), ejectCard() and retractCard()) and a required one named Card_Re which receives messages from its controller by the way of three methods.

The component Timer**.** As presented in Figure 10, this component has two interfaces. The provided one, Timer_Pr, offers two functionalities: it can be started with a fixed time and interrupted before the timeout is reached. When the timeout is reached, the timer sends this information through its required interface Timer_Re.

The component MultiLights**.** This component presented in Figure 11 is a light box that proposes several color lights. It offers, by the way of its provided interface MLight_Pr, the following functionalities: the chosen light can be turned on or turned off. When the light is turned off, one can choose a light color from predefined ones.

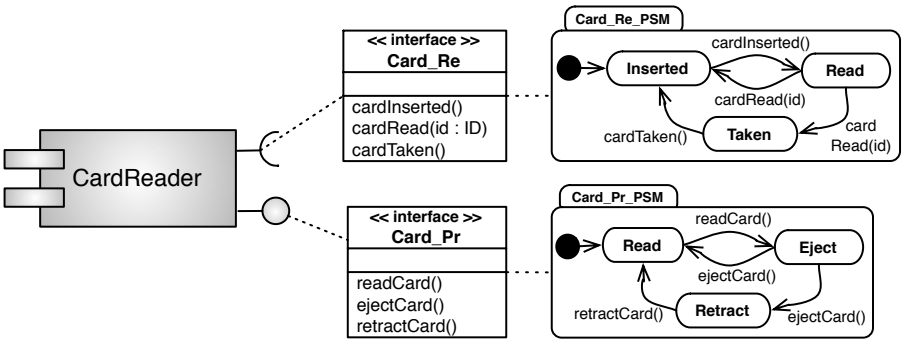


Figure 9: Component CardReader and its interfaces Card_Pr and Card_Re

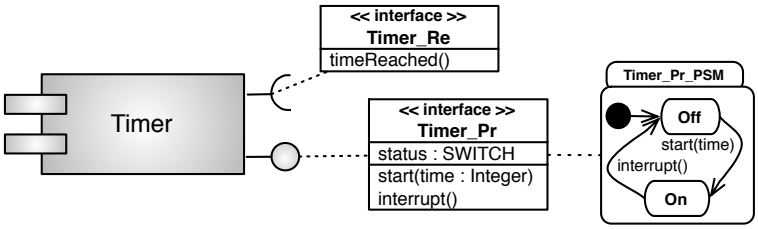


Figure 10: Component Timer and its interfaces Timer_Pr and Timer_Re

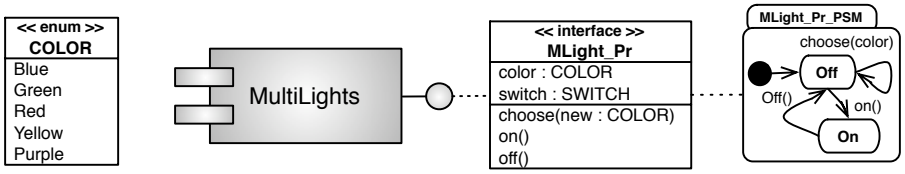


Figure 11: Component MultiLights and its provided interface MLight_Pr

4.2 The component Identification2

A component Identification2 can be defined by assembling these three existing components, as depicted in Figure 12 in order to fulfill the requirements. The required and provided interfaces Ident_Re and Ident_Pr of Identification2 have to be defined in terms of the components CardReader, Timer and MultiLights through their interfaces Card_Re, Card_Pr, Timer_Re, Timer_Pr and MLight_Pr.

A "new" component Controller is introduced to manage the interactions between all these interfaces. Identification2 *delegates* to Controller its interfaces Ident_Re and Ident_Pr:

- Controller realizes for Identification2 the interface Ident_Pr, and
- Controller uses through Identification2 the interface Ident_Re.

Figure 13 shows the sequence of operation calls between all the components to be assembled to produce Identification2 : there is the adaptation protocol between all the interfaces, that shows for all the required operations, the reaction in terms of provided operations calls.

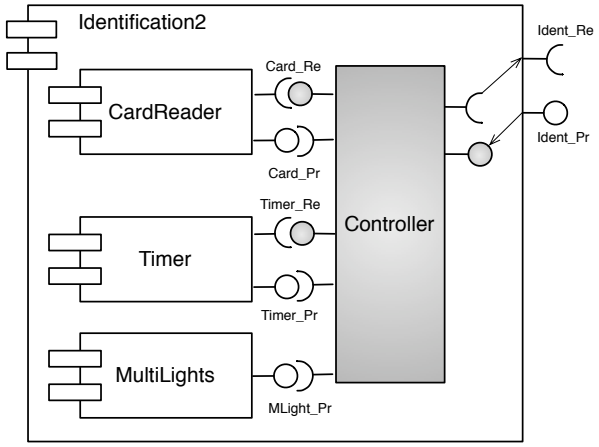


Figure 12: Architecture of the component Identification2

To prove that the assembly of CardReader, Timer and MultiLights through Controller is correct, we must prove that Controller:

- *realizes* the provided interface Ident_Pr delegated by Identification2, *realizes* the required interface Card_Re of CardReader, and *realizes* the required interface Timer_Re of Timer,
- *uses* the provided interfaces Card_Pr, Timer_Pr and MLight_Pr of the three existing components, and the required interface Ident_Re delegated by Identification2.

This UML 2.0 architecture can be expressed by the B architecture given in Figure 14 with two levels of refinement:

- the B abstract model, Controller_abs, which *extends* all the interfaces to be realized,
- the B refinement model, Controller, which
  - *includes* all the interfaces to be used, and
  - *refines* the abstract model Controller_abs.

The B refinement model of Controller is given in Figure 15:

- the available components are included, i.e. Ident_Re, Card_Pr, Timer_Re and MLight_Pr
- its gluing invariant expresses how to obtain the required attributes green and red from the attributes color and switch of the provided interfaces,
- the operations clause describes all the needed methods in terms of the used ones. The sequence diagram, given in Figure 13, which gives the protocol of the adaptation can help us to express these needed methods.

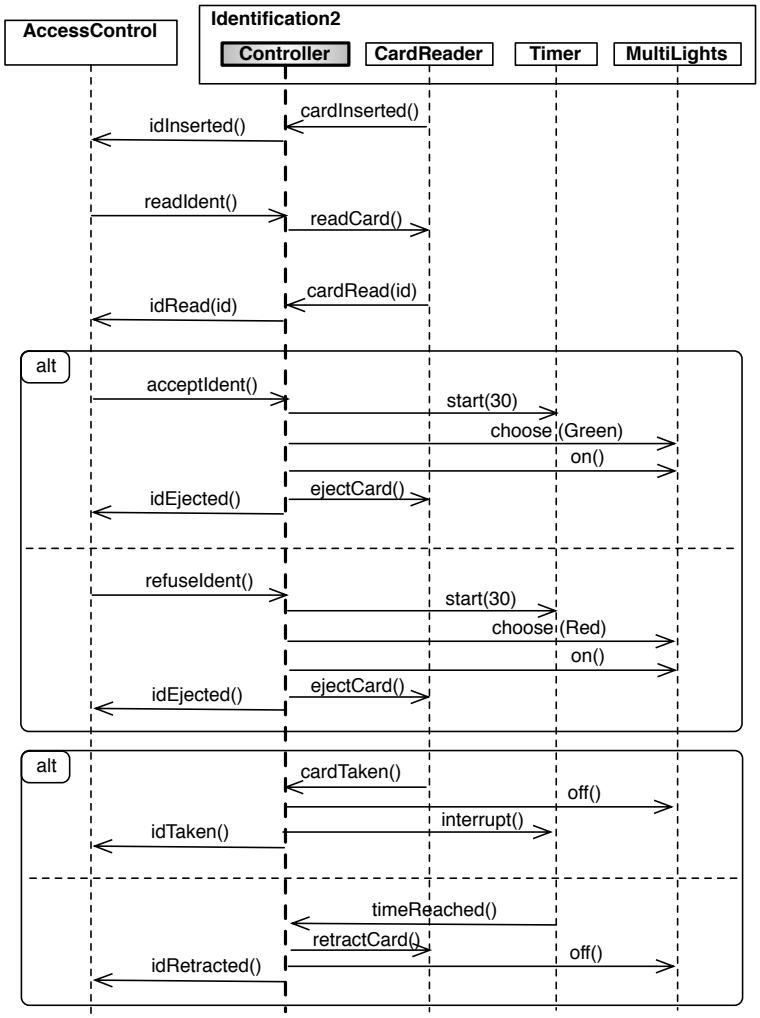


Figure 13: Sequence diagram for the Identification2 component

As an example, let us consider the needed method acceptIdent() of the Ident_Pr interface of Identification2. This method is called when an inserted card has been authorized to enter the building. The required result, as expressed in the sequence diagram of Figure 13 must be that a green light is turned on during a fixed time and the card is ejected. This requirement is expressed in the B operation acceptIdent by:

1. a timer is started: start(30),
2. the light's color is fixed to green if it is necessary (method choose(green)) before the light is turned on (method on),
3. the card is ejected, ejectCard, and
4. the environment is informed, isEjected.

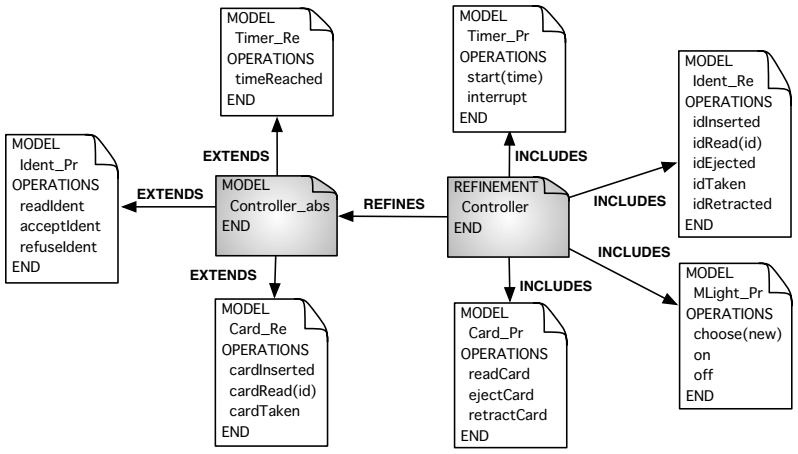


Figure 14: B architecture of the component Controller

```
REFINEMENT
   Controller
REFINES
   Controller_abs
SEES
   Type
INCLUDES
   Ident_Re, Card_Pr, Timer_Pr, MLight_Pr
INVARIANT
   ((color = Green ∧ switch = SW_On) ⇒ green = SW_On)
   ∧ (¬(color = Green ∧ switch = SW_On) ⇒ green = SW_Off)
   ∧ ((color = Red ∧ switch = SW_On) ⇒ red = SW_On)
   ∧ (¬(color = Red ∧ switch = SW_On) ⇒ red = SW_Off)
OPERATIONS
/* Ident_Pr */
readIdent =
   BEGIN
   readCard
   END ;
acceptIdent =
   BEGIN
   start(30) ;
   IF color = Green THEN on
   ELSE choose(Green) ; on
   END ;
   ejectCard ; idEjected
   END ;
refuseIdent =
   BEGIN
   start(30) ;
   IF color = Red THEN on
   ELSE choose(Red) ; on
   END ;
   ejectCard ; idEjected
   END ;
/* Timer_Re */
timeReached =
   BEGIN
   retractCard ; off ; idRetracted
   END ;
/* Card_Re */
cardInserted =
   BEGIN
   idInserted
   END ;
cardRead(uid) =
   BEGIN
   idRead(uid)
   END ;
cardTaken =
   BEGIN
   off ; interrupt ; idTaken
   END
END
```

Figure 15: B Model of the component Controller

We prove using B4free that the proposed component Controller is a correct implementation of the required functionalities in terms of the three existing components. With the B prover, we check

- that Controller refines all the required interfaces. This guarantees that the required behavioral and protocol aspects are preserved by the assembling. Of course, the signature level is also considered,

- the correctness of the use of the provided interfaces by the inclusion of their B interface models.

The example of adapters presented in this paper is a part of the case study of an access control system, as presented in section 2. The AccessControl system is equipped with other interfaces not presented in this paper as shown in Figure 1: DB_Re, to be connected with a database and Entry_Pr, Entry_Re, and Exit_Pr, to manage turnstiles. Existing components are used to answer the requirements of the AccessControl system. The Database component provides an interface Database_Pr to access to the stored informations and the Turnstile component has two interfaces Turn_Pr and Turn_Re to lock and unlock the turnstile. The access control system has been completely developped using our component-based approach. AccessControl must be connected to other components through specific adapters. In this paper, we have presented Adapter1 and Controller. Other adapters Entry1, Entry2, Exit1 and Database1 (decomposed into three steps of refinement to ease the proof, giving three versions of the adapter) are similarly defined. The table 1 gives an idea about all the proof obligations generated and discharged for the different components and adapters.

	Obvious POs	**POs**	**Interactives** POs
ID_Pr	11	0	0
ID_Re	7	0	0
Entry_Re	5	0	0
Exit_Pr	3	0	0
DB_Re	12	10	4
Ident_Pr	13	1	0
Ident_Re	11	0	0
Card_Pr	6	1	0
Card_Re	6	1	0
Timer_Pr	5	0	0
Timer_Re	0	0	0
MLight_Pr	11	0	0
Turn_Pr	5	0	0
Turn_Re	3	0	0
Database_Pr	3	0	0
Adapter1	12	5	0
Controller_abs	10	0	0
Controller	45	6	2
Entry1	9	2	0
Entry2	3	0	0
Exit1	3	0	0
Database1-1	6	2	2
Database1-2	6	2	2
Database1-3	5	8	2
TOTAL	**203**	**38**	**12**

Table 1: POs of AccessControl

5 Related Works

In an earlier paper [19], we have investigated 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. We have proposed a method consisting of four steps to guide this process.

Several proposals for verifying the interoperability between components have been made. In [15], Estevez and Fillottrani analyze how to apply algebraic specifications with refinement to component development, with a restriction to the use of modules that are described as class expressions in a formal specification language. They present several refinement steps for component development, introducing in each one design decisions and implementation details.

Our work focuses on the verification of interoperability of components through their interfaces using B assembling and refinement mechanisms.

Zaremski and Wing [36] propose an approach to compare two software components. They determine whether one required component can be substituted by another one. They use formal specifications to model the behavior of components and exploit the Larch prover to verify the specification matching of components.

In [9], a subset of the polyadic π-calculus is used to deal with the component interoperability at the protocol level. π-calculus is a well suited language for describing component interactions. Its main limitation is the low-level description of the used language and its minimalistic semantic. In [17, 18], protocols are specified using a temporal logic based approach, which leads to a rich specification for component interfaces. Henzinger and Alfaro [3] propose an approach allowing the verification of interfaces interoperability based on automata and game theories: this approach is well suited for checking the interface compatibility at the protocol level. In [6], the three levels of interface compatibilities are considered on web service interfaces described by transition systems.

Several proposals for component adaptation have already been made. The need of adaptation and assembly mechanisms was recognized in the late nineties [8, 14, 20]).

Some practice-oriented studies have been devoted to analyze different issues when one is faced with the adaptation of a third-party component [16]. A formal foundation to the notion of interoperability and component adaptation was set up in [35]. Component behavior specifications are given by finite state machines which are well known and support simple and efficient verification techniques for the protocol compatibility. Braccalia & al [7] specify an adapter as a set of correspondences between methods and parameters of the required and provided components. An adapter is formalized as a set of properties expressed in π-calculus. From this specification and from both interfaces, they generate a concrete implementable adapter.

Reussner and Schmit consider a certain class of protocol interoperability problems in the context of concurrent systems. For bridging component protocol incompatibilities, they generate adapters using interfaces described by finite state machines [29, 30].

Automatic generation of adapters is limited as one has to ensure the decidability of the interfaces inclusion problem, which is necessary to perform automated interoperability checks; one could only generate adapters for specific classes of assembly.

In our approach, we are not only concerned with specific classes of interoperability but with adapters in general. We propose to give general schemes to specify and verify adapters, not to generate them automatically.

6 Conclusion and Perspectives

The success of the component-based paradigm has received considerable attention in the software development field in industry and academia like in other engineering domains. We have presented an approach which contributes to specify component-based systems with high safety requirements. Our approach concerns the first steps of the system development life-cycle, from the requirements phase to the specification one, and aims at using existing languages and tools. We focus on the integration of components and the assembly mechanisms : components are considered as black-boxes described by their vis-

ible 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 expresses the mapping between required and provided variables and how required methods are implemented in terms of the provided ones. We have presented a general schema of assembly with both cases of interfaces.

The use of the B formal method and its refinement and assembling mechanisms to model the component interfaces and the adapters allows us to define rigorously the interoperability between components and to check it with support tools. The B prover guarantees that the adapter is a correct implementation of the required functionalities in terms of the existing components. Within this approach, the verification of the interoperability between the connected components is done at three levels, the signature, the semantic and the protocol levels.

We are currently working on extending this approach when additional abnormal behaviors are introduced to increase dependability of our component architecture and to preserve the normal cases. We have introduced two kinds of dependability mechanisms, one for security and one for safety. To extend the proposed approach, we study the definition of adapter schemas corresponding to different cases of component architecture. The idea is not to automatically generate the adapters, but to propose schemas to develop and verify the adaptation.

References

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

[2] AFADL'2000. Etude de cas : système de contrôle d'accès. In *Journées AFADL, Approches formelles dans l'assistance au développement de logiciels*, 2000. actes LSR/IMAG.

[3] L. Alfaro and T. A. Henzinger. Interface automata. In *9th Annual Aymposium on Foundations of Software Engineering, FSE*, pages 109–120. ACM Press, 2001.

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

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

[6] D. Beyer, A. Chakrabarti, and T. A. Henzinger. Web service interfaces. In *Proceedings of the 14th International World Wide Web Conference (WWW 2005)*, pages 148–159. ACM Press, 2005.

[7] A. Bracciali, A. Brogi, and C. Canal. A formal approach to component adaptation. In *Journal of Systems and Software*, volume 74, pages 45–54. Elsevier Science Inc., 2005.

[8] A. W. Brown and K. C. Wallnan. Engineering of component-based systems. In *Proceedings of the 2nd IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '96)*, page 414. IEEE Computer Society, 1996.

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

[10] C. Canal, J. M. Murillo, and P. Poizat. Software adaptation. *L'Objet*, 12(1):9–31, 2006.

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

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

[13] Clearsy. B4free, 2004. `http://www.b4free.com`.

[14] I. Crnkovic, S. Larsson, and M. Chaudron. Component-based development process and component lifecycle. In *27th International Conference Information Technology Interfaces (ITI)*. IEEE, 2005.

[15] E. Estevez and P. Fillottrani. Algebraic Specifications and Refinement for Component-Based Development using RAISE. *Journal of Computer Science and Technologie*, 2(7):28–33, 2002.

[16] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch: Why Reuse is so Hard. *IEEE Software*, 12(6):17–26, 1999.

[17] 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.

[18] J. Han. Temporal logic based specification of component interaction protocols. In *Proceedings of the Second Workshop on Object Interoperability ECOOP'2000*, pages 12–16. Springer-Verlag, 2000.

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

[20] G. Heineman and H. Ohlenbusch. An evaluation of component adaptation techniques. Technical Report WPI-CS-TR-98-20, Department of Computer Science, Worcester Polytechnic Institute, February 1999.

[21] M. Heisel, T. Santen, and J. Souquières. Toward a formal model of software components. In *Proc. 4th International Conference on Formal Engineering Methods - ICFEM'02*, number 2495 in LNCS, pages 57–68. Springer-Verlag, 2002.

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

[23] H. Ledang and J. Souquières. Contributions for modelling UML state-charts in B. In *Third International Conference on Integrated Formal Methods - IFM'2002*, pages 109–127, Turku, Finland, 2002.

[24] 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.

[25] Microsoft. *.Net*. `http://www.microsoft.com/net`.

[26] I. Mouakher, A. Lanoix, and J. Souquières. Component Adaptation: Specification and Verification. In *Proc. of the 11th Int. Workshop on Component Oriented Programming (WCOP'06), satellite workshop of ECOOP 2006*, pages 23–30, 2006.

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

[28] Object Mangagement Group (OMG). *Corba Component Model Specification*, 2006. version 4.0.

[29] R. H. Reussner, H. W. Schmidt, and I. H. Poernomo. Reasoning on software architectures with contractually specified components. In A. Cechich, M. Piattini, and A. Vallecillo, editors, *Component-Based Software Quality: Methods and Techniques*, pages 287–325. 2003.

[30] H. W. Schmidt and R. H. Reussner. Generating adapters fo concurrent component protocol synchronisation. In I. Crnkovic, S. Larsson, and J. Stafford, editors, *Proceeding of the 5th IFIP International conference on Formal Methods for Open Object-based Distributed Systems*, pages 213–229, 2002.

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

[32] Sun Microsystems. *JSR 220: Enterprise JavaBeans*, 2006. Version 3.0, Final Realase.

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

[34] W3C. *Web Services*. `http://www.w3.org/2002/ws`.

[35] D. D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.

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