Component-based Development using the B method

Arnaud Lanoix, Jeanine Souquières

Component-based Development using the B method

Arnaud Lanoix and Jeanine Souquières

LORIA – Nancy-Université, CNRS
Campus scientifique
F-54506 Vandoeuvre-Lès-Nancy
{lanoix, souquier}@loria.fr

Abstract. In component-based software development approaches, components are considered as black boxes. Components communicate through required and provided interfaces which describe their visible behaviors. In the best cases, the provided interfaces are checked compatible with the corresponding required interfaces, but in general cases, adapters have to be introduced to connect them. Compatibility between required and provided interfaces concerns the interface signatures, behavioral aspects and protocol level. We propose to specify component interfaces in B in order to verify these three levels of interoperability. The use of B assembling and refinement mechanisms eases the verification of the interoperability between interfaces and the correctness of the component assembly. The verification is done by the B prover.
Keywords: component-based development, provided interface, required interface, interoperability, compatibility, adaptation, assembly

1 Introduction

Recent works have shown that assembling components independently produced [1] and taking into account the verification of their assembly with appropriate tools is a promising approach. The underlying idea is to develop software systems by assembling existing parts, as it is common in other engineering disciplines. Among the advantages of such an approach, we can cite: (i) reusability of trustworthy software components, (ii) reduction of the costs of development due to the reusability, and (iii) flexibility of systems developed by this approach. But this approach is not currently well supported by standard design methods nor adapted to critical applications. On the one hand, current technologies of components design 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 are designed by examining existing components (potentially off-the-shelf) to see how they meet the expected requirements and decide how they can be integrated to provide the expected functionalities. Finally, the system is engineered

by assembling the selected components with some locally developed pieces of code.

Components are black-boxes units of composition which contractually 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 the component informations and functionalities. The offered services by a component are described by provided interfaces and the needed services are described by required interfaces.

For different components independently developed, to be deployed and to work together, they must interoperate, i.e. 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. The idea to define component interfaces using B has been introduced in an earlier paper [2]. Semantics and protocols of the component services can be easily modeled by the B formalism. The use of the B refinement [3] to prove that two components are compatible at the signature and semantics levels has been explored in [4]. To guarantee interoperability of components, each connection of a required interface with another provided interface has to be considered. In the best cases, a provided interface – after some renaming – constitutes an implementation of the required interface. In the general cases, to construct a working system out of components, adapters have to be defined, connecting the required methods and attributes to the provided ones [5]. An adapter is seen as 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 and at the behavioral and protocol levels, it expresses how the required methods are implemented in terms of the provided ones.

In this paper, we generalize the previous results, taking into account more general situations with the use of both cases of interfaces for different components to be connected. The use of the B method and tools allows us to give 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 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 [6]. Persons who are authorized to enter the building have an access card with some identification information stored on it. Turnstiles block the entrance and the exit of each building until an authorization is given. We will focus on the Identification component which is composed of a card reader and two lights, a green light and a red light. The lights indicate if the authorization has been accepted (the green light turns on during a fixed time of 30 seconds) or denied (the red light turns on during a fixed time of 30 seconds). The two lights cannot be turned on at the same time. The card is ejected and must be taken by the person before the light turns off. If the person does not take the card within some time limit, the card is retracted and kept by the system.

2.1 Overall architecture of the system

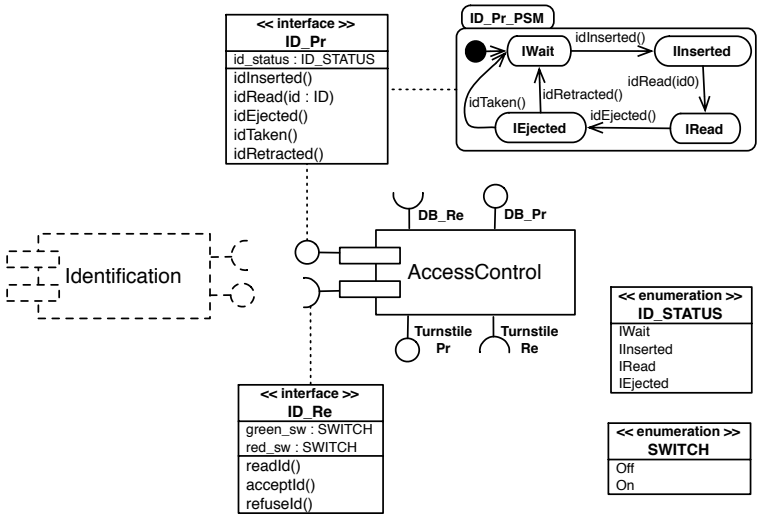
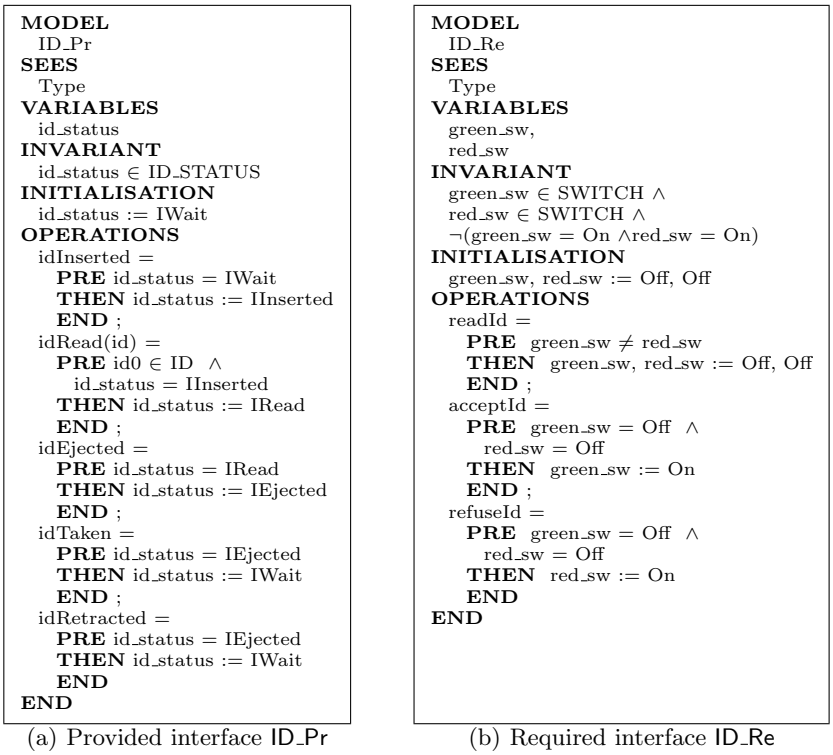


Fig. 1. Partial view of the architecture of the access control system

A partial view of the architecture of the access control system is given Figure 1 as a UML 2.0 composite structure diagram [7]. Such diagrams contain named rectangles corresponding to the components of the system; here, we have depicted two components, AccessControl corresponding to the system requirements and Identification component, with a dotted line box, to denoted that it is the component we want to define. Components are connected by means of

interfaces which may be required or provided. Required interfaces explicit 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.

Two interfaces have been outlined in Figure 1, modeled by class diagrams with their different attributes and methods. ID_Pr corresponds to the provided interface of AccessControl related to the Identification component and ID_Re to its required interface. Enumerated data types are defined using the stereotype "enumeration". As an example of an interface specification, the usage protocol of the ID_Pr provided interface is modeled by a Protocol State Machine (PSM). This PSM specifies the order of the allowed method calls of ID_Pr from its initial state.



(a) Provided interface ID_Pr (b) Required interface ID_Re

Fig. 2. B Models for the interfaces of AccessControl

B models for the provided and required interfaces of AccessControl are given Figure 2:

- the ID_Pr B model reflects the ID_Pr_PSM. The variable id_status has four possible states and its initial state is IWait. After an idInserted() call, its state is changed to IInserted, that is reflected in the B model,
- the ID_Re B model expresses the required behaviors about the lights. Two variables green_sw and red_sw reflects the lights state. The green light must be turned on if the authorization has been accepted (acceptId), otherwise the red light must be turned on (refuseId). An invariance property expresses that the two lights cannot be turned on at the same time.

It is to be noted that B models can be obtained from UML 2.0 diagrams by applying systematic derivation rules [8,9]. In this paper, we consider that the B models are given with the interface description of each component.

2.2 Existing components

We dispose of three existing components, namely CardReader, Timer and MultiLights. Their functionalities are known by their interface descriptions in UML 2.0 associated to B models for behavioral and protocol specifications.

The component CardReader**.** This component reads information from an access card. It is equipped with two interfaces, as presented Figure 3, a provided one named Card_Pr with three methods (read(), eject() and retract()) and a required one named Card_Re which gives information to its controller by the way of five methods.

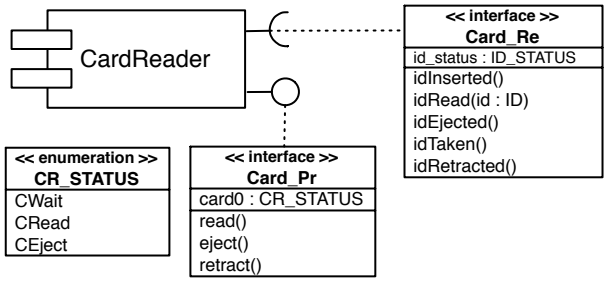


Fig. 3. Component CardReader and its interfaces Card_Pr and Card_Re

The component Timer**.** As presented Figure 4, 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.

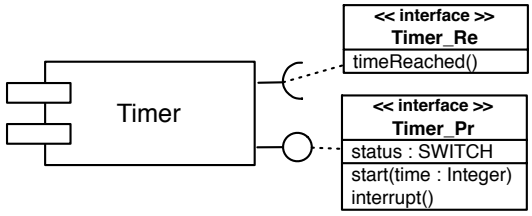


Fig. 4. Component Timer and its interfaces Timer_Pr and Timer_Re

The component MultiLights**.** This component presented Figure 5 is a light box that proposes several color lights. It offers, by the way of its provided interface MLight_Pr, the next functionalities: the chosen light can be turned on and turned off. When the light is turned off, one can choose a light color from predefined ones.

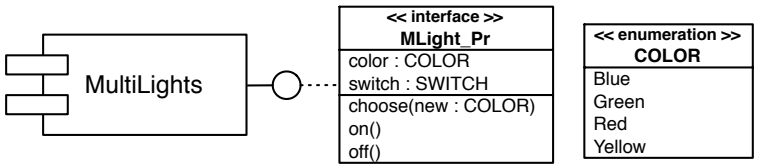


Fig. 5. Component MultiLights and its provided interface MLight_Pr

2.3 The component Identification

The component Identification can be defined by a composition of the three existing components, namely CardReader, Timer and MultiLights. In order to be connected to the access control system, it requires an interface compatible with ID_Pr and provides another interface compatible with ID_Re. Its architecture is depicted Figure 6:

- the provided interface ID_Pr of AccessControl can be directly connected to the required interface of the component CardReader, Card_Re,
- the required functionalities of ID_Re have to be defined by using the components CardReader, Timer and MultiLights through their interfaces Card_Pr, Timer_Re, Timer_Pr and MLight_Pr. To manage the interactions between all these interfaces, a component Assembly is introduced.

To prove the correctness of the proposed assembly, we must prove that:

- the required interface Card_Re of CardReader is *compatible* with the provided interface ID_Pr of AccessControl,

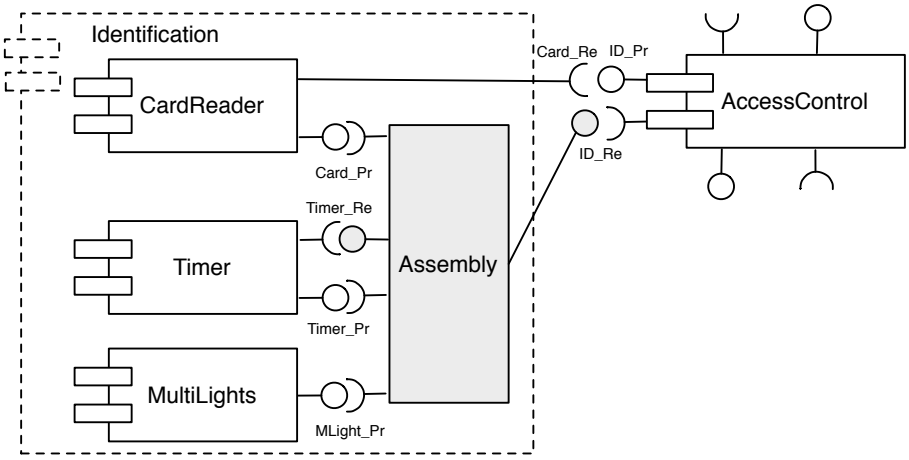


Fig. 6. Architecture of the component Identification

– the *assembly* of the provided interfaces Card_Pr, Timer_Pr and MLight_Pr
  - *provides* the required interface ID_Re of AccessControl,
  - *provides* the required interface Timer_Re of Timer.

3 Simple Case of Trustworthy Component Assembly

Interoperability means the ability of components to communicate and cooperate despite differences in their implementation language, their execution environment, or their model abstraction [10]. Two components are interoperable if all their interfaces are compatible [4]. More precisely, for each required interface of a considered component, there exist 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 exist 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 B notation has been used in an earlier paper to define component interfaces [2] because its underlying concepts of machine and refinement fit well with

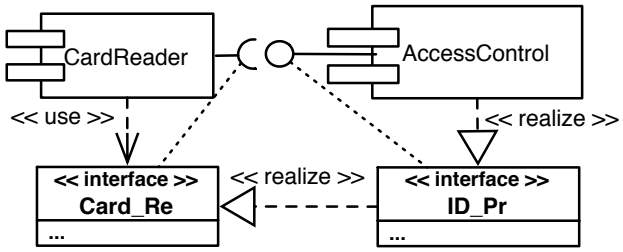


Fig. 7. Compatibility between Card_Re and ID_Pr

components interoperability [4] and the method is equipped with powerful tool support [11,12].

Figure 7 presents a visualization, using a UML composite diagram, of the compatibility problem between the required interface Card_Re of the component CardReader and the provided interface ID_Pr of the component AccessControl:

ID_Pr *realizes* Card_Re

Using the B method, we have to prove that the B model of ID_Pr is a correct refinement of the B model of Card_Re. To do this, we propose the next schema, modeled by a B refinement machine, named Connector as presented Figure 8 which:

- *refines* the B model of the required interface Card_Re,
- *extends* the B model of the provided interface Id_Pr.

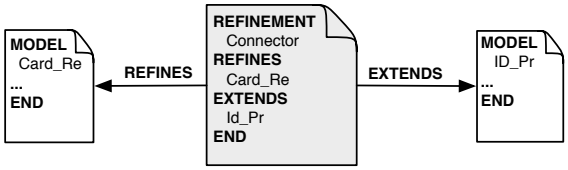


Fig. 8. B schema of the connection between Card_Re and ID_Pr

The methods of the provided interface implements directly the methods of the required interface. The B *extends* assembly mechanism corresponds to the inclusion of the model where each operation of the included machine is promoted [13].

The proof of the refinement is obvious. That means that the required interface Card_Re of the component CardReader is compatible with the provided interface ID_Pr of AccessControl. The interoperability is verified at the signature, semantic and protocol levels. As a consequence, Connector implements Card_Re in terms of ID_Pr.

In the general case, to construct a working system 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 and, at the behavioral and protocol levels, it expresses how the required operations are implemented in terms of the provided ones. In [5], we have study 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.

4 General Case of Component Assembly

The general case of component assembly concerns the use of both type of interfaces for different components to be connected. We define a new specific component that manages all the considered components through their required and provided interfaces. This assembling component realizes all the required interfaces of the considered components using their provided interfaces.

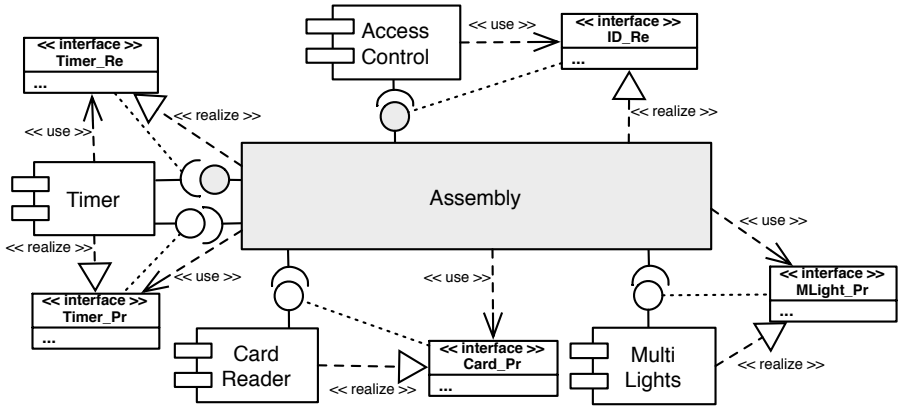


Fig. 9. UML composite diagram of Assembly

A UML architecture of this assembling component is given Figure 9 for the running example. The different interactions with the three components introduced previously, CardReader, Timer and MultiLights are outlined in order to fulfill the required interface ID_Re of AccessControl and Timer_Re of Timer. The component Assembly:

- *realizes* the required interfaces ID_Re of AccessControl,
- *realizes* the required interfaces Timer_Re of Timer,
- *using* the provided interfaces Card_Pr, Timer_Pr and MLight_Pr of the three existing components.

A B architecture of the component Assembly is given Figure 10 with two levels of B models:

- the B abstract model, Assembly_abs, which *extends* all the required interfaces,
- the B refinement model, Assembly, which
  - *includes* all the provided interfaces and
  - *refines* the abstract model Assembly_abs.

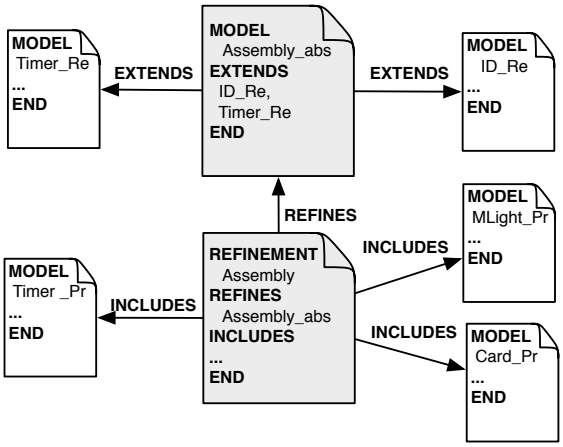


Fig. 10. B architecture of the component Assembly

For the running example, the B refinement model Assembly is given Figure 11:

- the three available components are included,
- its gluing invariant expresses how to obtain the required attributes green_sw and red_sw from the attributes of the provided interfaces,
- the operations clause describes all the required methods in terms of the provided ones.

Let us consider the method acceptId(). It is called by AccessControl when the card has been authorized to enter the building. The required result must be that a green light is turned on and the card is ejected. The method acceptId() is enabled only if the card reader had read the card before (reader.card0 = CRead). First, a timer is started (timer.start(30)). Next, the color's light is fixed to green (lights.choose(col)) before the light is turned on (lights.on). Finally the card is ejected (reader.eject).

The B prover generates 168 obvious proof obligations and 30 "real" proof obligations which are automatically proved. We conclude that the proposed Assembly component is a correct implementation of the required functionalities in terms of the three existing components. With the B prover, we check

```
REFINEMENT
  Assembly
REFINES
  Assembly_abs
SEES
  Type
INCLUDES
  reader.card_Pr,
  lights.MLight_Pr,
  timer.Timer_Pr
INVARIANT
  (lights.color = Green ∧
    lights.switch = On ∧
    timer.status = On ∧
    reader.card ∈ {Cwait,CEject}
  ⇒ green_sw = On ∧ red_sw = Off)
  ∧ (lights.color = Red ∧
    lights.switch = On ∧
    timer.status = On ∧
    reader.card ∈ {Cwait,CEject}
  ⇒ green_sw = Off ∧ red_sw = On)
OPERATIONS
  timeReached =
    BEGIN
    IF lights.switch = On
    THEN lights.off
    END ;
    IF reader.card0 = CEject
    THEN reader.retract
    END
    END ;
  readId =
    SELECT reader.card0 = CWait
    THEN
      IF timer.status = On
      THEN timer. interrupt
      END ;
      IF lights.switch = On
      THEN lights.off
      END ;
      reader.read
    END ;
  acceptId =
    SELECT reader.card0 = CRead
    THEN
      IF timer.status = On
      THEN timer.interrupt
      END ;
      timer.start(30) ;
      IF lights.switch = On
      THEN lights.off
      END ;
      IF lights.color ≠Green
      THEN
        LET col BE col = Green
        IN lights.choose(col)
        END
      END ;
      lights.on ;
      reader.eject
    END ;
  refuseId =
    SELECT reader.card0 = CRead
    THEN
      IF timer.status = On
      THEN timer.interrupt
      END ;
      timer.start(30) ;
      IF lights.switch = On
      THEN lights.off
      END ;
      IF lights.color ≠Red
      THEN
        LET col BE col = Red
        IN lights.choose(col)
        END
      END ;
      lights.on ;
      reader.eject
    END
END
```

Fig. 11. B Model of the component Assembly

- that Assembly 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.

5 Related Work

Different works have addressed the composition of B specifications. B assembly mechanisms such as *extends* and *includes* used in this work have been introduced in [3] and defined in terms of component primitives in order to perform altogether incremental machine construction and proofs of consistency of the elaborated machines [13]. The extraction of understandable B models from a monolithic one combined with the refinement mechanism is presented in [14]. In [15], J.–R. Abrial gives a methodology of decomposition of a B model into sub-models to refine separately each sub-model before recomposing them at the end of the refinement process. In [16], the B method is extended to allow a parallel composition of B models using shared variables to be taken into account.

Butler [17], Treharne and Schneider [18,19] combine B with CSP. They associate to each B model, a "controller" expressed in CSP, which describes the communication with other B models. In [20], the composition of B models is expressed by a set of synchronization constraints on the operations.

In [21], 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 [22,23] 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 [24], 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 [25,26], protocols are specified using a temporal logic based approach, which leads to a rich specification for component interfaces. Henzinger and Alfaro [27] 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 [28], the three levels of interface compatibilities are considered on web service interfaces described by transition systems.

Several proposals for component adaptation have already been done. The need of adaptation and assembly mechanisms was recognized in the late nineties [29,30] (see also [31]).

Some practice-oriented studies have been devoted to analyze different issues when one is faced to the adaptation of a third-party component [32]. A formal foundation to the notion of interoperability and component adaptation was set up in [10]. 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 [33,34] 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 interface described by finite parameterized state machines [35,36,37].

Automatic generation of adapters are limited as one had 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 interoperability.

In our approach, we are not only concerned with specific classes of interoperability but with adapters in general. We propose to give general schema to specify and verify adapters, not to generate them automatically. We have proposed a method consisting of four steps to guide this process [38].

6 Conclusion and Perspectives

We have presented an approach for component-based system and software development. Components are considered as black-boxes described by their required and provided interfaces. To construct a working system out of existing components, adapters have to be defined. In this paper, we focus on the component Assembly which is a new component introduced to manage interactions between different components to be connected, with both cases of interfaces.

The component Assembly realizes all the required interfaces using its provided ones. It is defined as a B refinement, including the models of the provided interfaces. The B prover guarantees that this component is a correct implementation of the required functionalities in terms of the existing components. This approach allows us to verify the interoperability between the connected components at the three levels: signature, semantic and protocol levels.

We are currently working on alternative versions of compatibility and their mappings to the B refinement in order to give patterns for the assembly of components.

References

1. Szyperski, C.: Component Software. ACM Press, Addison-Wesley (1999)
2. Chouali, S., Souquières, J.: Verifying the compatibility of component interfaces using the B formal method. In: International Conference on Software Engineering Research and Practice. (2005)
3. Abrial, J.R.: The B Book. Cambridge University Press (1996)
4. Chouali, S., Heisel, M., Souquières, J.: Proving Component Interoperability with B Refinement. In Arabnia, H.R., Reza, H., eds.: International Worshop on Formal Aspects on Component Software, CSREA Press (2005) 915–920 To appear in ENCTS 2006.
5. Mouakher, I., Lanoix, A., Souquières, J.: Component Adaptation: Specification and Verification. In: Proc. of the 11th Int. Workshop on Component Oriented Programming (WCOP 2006). (2006) 23–30
6. Afadl2000: Etude de cas : système de contrôle d'accès. In: Journées AFADL, Approches formelles dans l'assistance au développement de logiciels. (2000) actes LSR/IMAG.
7. Object Management Group: UML superstructure specification, v2.0 (2005)
8. Meyer, E., Souquières, J.: A systematic approach to transform OMT diagrams to a B specification. In: Proceedings of the Formal Method Conference. LNCS 1708, Springer-Verlag (1999) 875–895
9. Ledang, H., Souquières, J.: Contributions for modelling UML state-charts in B. In: Third International Conference on Integrated Formal Methods - IFM'2002, Turku, Finland (2002)
10. Yellin, D.D.M., Strom, R.E.: Protocol specifications and component adaptors. ACM Transactions on Programming Languages and Systems **19**(2) (1997) 292–333
11. Steria – Technologies de l'information: Obligations de preuve: Manuel de référence, version 3.0. (1998)
12. Clearsy: B4free. Available at *http://www.b4free.com* (2004)
13. Bert, D., Potet, M.L., Rouzaud, Y.: A study on components and assembly primitives in B. In: Proceedings of 1st Conference on the B method. (1996) 47–62
14. Bontron, P., Potet, M.: Automatic construction of validated B components from structured developments. In Bowen, J.P., Dunne, S., Galloway, A., King, S., eds.: ZB2000: Formal Specification and Development in Z and B. Volume 1878 of LNCS., Springer-Verlag (2000) 127–147
15. Abrial, J.R.: Discrete system models. Version 1.1 (2002)
16. Attiogbé, J.: Communicating B abstract systems. Research Report RR-IRIN 02.08 (2002) updated july 2003.
17. Butler, M.J.: CSP2B: A practical approach to combining CSP and B. Formal Aspects of Computing **12** (2000) 182–198
18. Treharne, H., Schneider, S.: Using a process algebra to control B OPERATIONS. In: 1st International Conference on Integrated Formal Methods (IFM'99), York, Springer Verlag (1999) 437–457
19. Schneider, S., Treharne, H.: Communicating *B* machines. In Bert, D., Bowen, J.P., Henson, M.C., Robinson, K., eds.: Formal specification and development in Z and B (ZB 2002). Volume 2272 of LNCS., Springer Verlag (2002) 416–435
20. Bellegarde, F., Julliand, J., Kouchnarenko, O.: Synchronized parallel composition of event systems in B. In Bert, D., Bowen, J.P., Henson, M.C., Robinson, K., eds.: Formal specification and development in Z and B (ZB'2002). Volume 2272 of LNCS., Springer-Verlag (2002) 436–457

21. Estevez, E., Fillottrani, P.: Algebraic Specifications and Refinement for Component-Based Development using RAISE. Journal of Computer Science and Technologie **2**(7) (2002)
22. Zaremski, A.M., Wing, J.M.: Signature matching: a tool for using software libraries. ACM Transactions on Software Engineering and Methodology **4**(2) (1995) 146–170
23. Zaremski, A.M., Wing, J.M.: Specification matching of software components. ACM Transaction on Software Engeniering Methodolology **6**(4) (1997) 333–369
24. Canal, C., Fuentes, L., Pimentel, E., Troya, J.M., Vallecillo, A.: Extending CORBA interfaces with protocols. Comput. J. **44**(5) (2001) 448–462
25. Han, J.: A comprehensive interface definition framework for software components. In: The 1998 Asia Pacific software engineering conference, IEEE Computer Society (1998) 110–117
26. Han, J.: Temporal logic based specification of component interaction protocols. In: Proceedings of the Second Workshop on Object Interoperability ECOOP'2000, Springer-Verlag (2000) 12–16
27. Alfaro, L., Henzinger, T.A.: Interface automata. In: 9 th Annual Aymposium on Foundations of Software Engineering, FSE, ACM Press (2001) 109–120
28. Beyer, D., Chakrabarti, A., Henzinger, T.A.: Web service interfaces. In: Proceedings of the 14th International World Wide Web Conference (WWW 2005), ACM Press (2005) 148–159
29. Brown, A.W., Wallnan, K.C.: Engineering of component-based systems. In: ICECCS '96: Proceedings of the 2nd IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '96), IEEE Computer Society (1996) 414
30. Heineman, G., Ohlenbusch, H.: An evaluation of component adaptation techniques. Technical Report WPI-CS-TR-98-20, Department of Computer Science, Worcester Polytechnic Institute (1999)
31. Crnkovic, I., Larsson, S., Chaudron, M.: Component-based development process and component lifecycle. In: 27th International Conference Information Technology Interfaces (ITI), IEEE (2005)
32. garlan, D., Allen, R., Ockerbloom, J.: Architectural Mismatch: Why Reuse is so Hard. IEEE Software **12**(6) (1999) 17–26
33. Braccalia, A., Brogi, A., Turini, F.: Coordinating Interaction Patterns. In Press, A., ed.: Symposium on Applied Computing (SAC'2001). (2001)
34. Bracciali, A., Brogi, A., Canal, C.: A formal approach to component adaptation. In: Journal of Systems and Software. (2005)
35. Reussner, R.H.: Adapting Components and Predicting Architectural Properties with Parameterised Contracts. In Goerigk, W., ed.: Tagungsband des Arbeitstreffens der GI Fachgruppen 2.1.4 und 2.1.9, Bad Honnef. (2001) 33–43
36. Schmidt, H.W., Reussner, R.H.: Generating adapters fo concurrent component protocol synchronisation. In Crnkovic, I., Larsson, S., Stafford, J., eds.: Proceeding of the 5th IFIP International conference on Formal Methods for Open Object-based Distributed Systems. (2002)
37. Reussner, R.H., Schmidt, H.W., Poernomo, I.H.: Reasoning on software architectures with contractually specified components. In Cechich, A., Piattini, M., Vallecillo, A., eds.: Component-Based Software Quality: Methods and Techniques. (2003)
38. Hatebur, D., Heisel, M., Souquières, J.: A Method for Component-Based Software and System Development. In Society, I.C., ed.: Proceedings of the 32 nd Euromicro Conference on Software Engineering And Advanced Applications (CBSE). (2006) To appear, August.