Component Adaptation: Specification and Verification

Inès Mouakher, Arnaud Lanoix, Jeanine Souquières

Component Adaptation: Specification and Verification

Inès Mouakher Arnaud Lanoix Jeanine Souquières

LORIA – CNRS – Université Nancy 2
Campus scientifique
F-54506 Vandoeuvre-Lès-Nancy
{mouakher, lanoix, souquier}@loria.fr

Abstract

In a component-based software development, components are considered as black boxes. They are only described by their interfaces expressing their visible behaviors. They must be connected in an appropriate way, through required and provided interfaces. To guarantee interoperability of components, we must consider each connection of a required interface with another provided interface. 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. They connect the required operations and attributes to the required ones. The interoperability between a required interface and provided interfaces through an adapter is guaranteed by the use of the B formal method with its underlying concept of refinement, and its powerful tool support, the B prover.

1 Introduction

The idea underlying the paradigm of component orientation [12, 25] is to develop software systems not from scratch but by assembling pre-fabricated parts, as it is common in other engineering disciplines. As in object orientation, components are encapsulated, and their services are only accessible via interfaces and their operations. 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 system to be implemented is achieved. A component is a unit of composition with contractually specified interfaces and explicit dependencies. An interface describes the services offered or required by a component without disclosing the component implementation. It is the only access to the informations of a component. The offered services by a component are described by a provided interface and the needed services are described by a required interface.

The success of applying the component based approach depends on the interoperability of the connected components. The interoperability can be defined as the ability of two or more entities to communicate and cooperate despite differences in their implementation language, their execution environment, or their model abstraction [13, 26]. The interoperability of two components concerns the compatibility between the required interface of one of the considered components with the provided interface of the other one. More precisely, three levels of interoperability have to be considered. The syntactic level covers static aspects of components interoperability. It concerns the interface signature: each attribute of the required interface must have a counterpart in the provided interface, but not necessarily vice versa; for each operation of the required interface, there exists an operation of the provided interface, such as their signatures are compatibles. The semantic level covers the behavioral aspects of components interoperability. The protocol level deals with the allowed sequences of methods calls that a component expects.

The specification of interfaces plays an important role in the verification of their compatibility. Most current interface modelling languages (IDLs), used in several component oriented platforms like JavaBeans [23, 24], CORBA [18], or COM [16], are limited for expressing signature (operation names, types, parameters) informations. They provide an insufficient information about component behaviors. Hence, one cannot insure trust in component based systems.

The availability of formal languages and tool support for specifying these interfaces 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 [8]. The use of the B refinement [1] to prove that two components are compatible at the signature and semantics levels has been explored in [7].

In this paper we focus on the generation of adapters that realize the matching between two or more existing components specified by their required and provided interfaces. We propose to specify interfaces in terms of UML 2.0 diagrams [17]. These diagrams are then automatically transformed into B specifications [15, 14]. A model of a correct adapter is generated in B. The verification of the interoperability is automatically done by the B prover: the B model of the

adapter is a refinement of the B model of the required interface using provided components. This verification process is done at the signature, semantic and protocol levels.

The rest of the paper is organized as follows. In Section 2, we give an overview of our component-based development specification with the specification of the component interfaces. We then propose the definition of adapters in Section 3 and the verification of the interoperability between the connected components. The case study of a simple access control system serves to illustrate our proposition. We discuss related work in Section 4. The paper finishes with some concluding remarks in Section 5.

2 Component-Based Development and Interface Specification

Our goal is to provide an approach for component-based software development that pays special attention to the question of how the interoperability between different components can be guaranteed. Components are specified 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. In this framework, adaptation between two components is a hard problem which has to be seen in an abstract way. We propose a methodology for specifying the required adaptation between two or more existing components by introducing a third component called Adapter. This new component is in charge, when possible, of mediating the interactions of the different components so that they can successfully interoperate.

The overall architecture of the system is expressed by a UML 2.0 composite structure diagram [17]. Such diagrams contain named rectangles corresponding to the components of the system. 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.

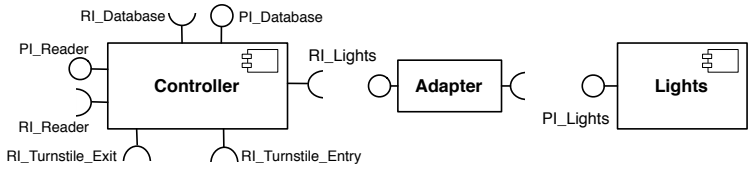

Figure 1: A partial view of the architecture of the access control system

Figure 1 presents a partial view of the architecture of the access control system where the access of authorized persons to a building is to be controlled. Persons have at their disposal access cards with identification information stored on it. There are two turnstiles, one at the entrance to the building, and one at the exit. At the entrance, there is also a card reader as well as a red and a green light. In the sequel, we will focus on the interaction between the Controller and the Lights components. The given requirement says that if the access is authorized, a green light is turned on, whereas, if the access is refused, a red light is turned on. More precisely, the green and the red lights cannot be turned on at the same time. The Controller component has several interfaces; one of its requested interface is related to the Lights component, namely RI_Lights. The Lights component has only one interface which is provided to a controller component, namely PI_Lights.

An intermediate component named Adapter has to be introduced: it is in charge to implement the links between the required interface of the Controller component and the provided interface of an existing Lights component.

2.1 Specifying Components

For each component of the architecture, a specification of each interface has to be set up [7]. A component interface specification consists of a data model described by a class diagram with its different attributes and operations. The usage protocol of the interface is modeled by a Protocol State Machine (PSM): for each operation, its pre- and post-conditions are specified.

2.1.1 The Controller Component

With respect to its interaction with a Lights component, the Controller component requires an interface RI_Lights as presented Figure 2. A PSM is associated to this interface to specify its externally visible behavior, i.e. its usage protocol. Safety constraints on the required interface can be added by the way of an invariant expressed by an OCL annotation.

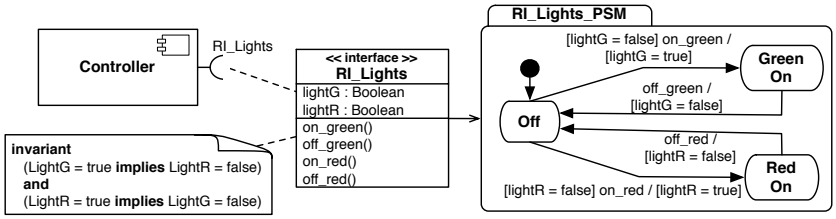

Figure 2: The component Controller with its interface RI_Lights and its associated PSM

Different components corresponding to the behavior of the Lights component used in Figure 1 are available in the component library. Let us consider two of them, one called MultiLights corresponding to a component with the possibility of choosing its color and a second one called SingleLight which is a simple light.

2.1.2 The MultiLights Component

The available MultiLights component offers, by the way of its provided interface called PI_MLights, the next functionalities: the light can be turned on and turned off. When the light is turned off, one can choose the light color from four predefined colors: blue, green, red and yellow. The UML specification of this component is given Figure 3, i.e. its provided interface, PI_MLights, and its associated PSM.

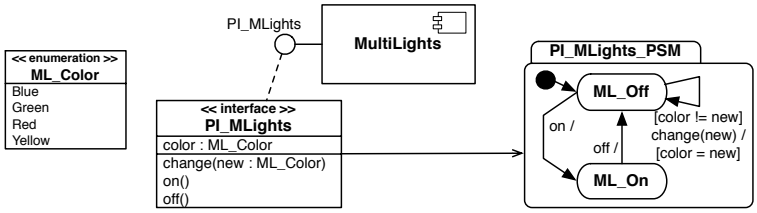

Figure 3: The MultiLights component with its interface PI_MLights and its associated PSM

2.1.3 The SingleLight component

Another component named SingleLight is available in the component library. It corresponds to a simple light which can only be turned on and turned off. Figure 4 gives its UML specification, i.e. its provided interface PI_SLight and its associated PSM.

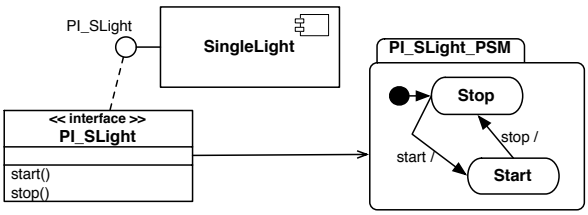

Figure 4: The SingleLight component with its interface PI_SLight and its associated PSM

2.2 Derivation of UML Interface Specifications to B

UML 2.0 proposes expressive graphical notations to specify components and their interfaces. Nevertheless, it does not provide a suitable framework neither to support formal verifications nor to check the interoperability. The B formal method [1] supports an incremental development process, using refinement. Proofs of invariance and refinement are part of each development. The proof obligations are generated automatically by support tools such as AtelierB [22] or B4free [9], an academic version of AtelierB.

Inspired from the derivation rules from UML 1.X class diagrams and state diagrams to B specifications [15, 14], we automatically translate the UML 2.0 interfaces and their associated PSMs into B specifications for checking their interoperability. Intuitively:

- a B model is derived from the interface,

- a set containing the "control" states of the associated PSM is added to the B model,
- each transition of the PSM is formalized by a B operation: pre- and post-conditions from a transition become preconditions and substitutions into the B model,
- new preconditions (and substitutions) about the "control" states are incorporated into each B operation to model the PSM.

Figure 5 gives the B model of the three component interfaces previously specified with UML.

```
MODEL
   RI_Lights
SETS
   RI_Lights_STATES = {Off, GreenOn, RedOn}
VARIABLES
   lightG, lightR, lr_state
INVARIANT
   lightG ∈ BOOL ∧
   lightR ∈ BOOL ∧
   lr_state ∈ RI_Lights_STATES ∧
   (lightG = TRUE ⇒ lightR = FALSE) ∧
   (lightR = TRUE ⇒ lightG = FALSE)
INITIALISATION
   lightG, lightR, lr_state := FALSE, FALSE, Off
OPERATIONS
   on_green =
      PRE lightG = FALSE ∧ lr_state = Off
      THEN lightG := TRUE || lr_state := GreenOn
      END ;
   off_green =
      PRE lr_state = GreenOn
      THEN  lightG := FALSE || lr_state := Off
      END ;
   on_red =
      PRE lightR = FALSE ∧ lr_state = Off
      THEN lightR := TRUE || lr_state := RedOn
      END ;
   off_red =
      PRE lr_state = RedOn
      THEN  lightR := FALSE || lr_state := Off
      END
END
```

(a) RI_Lights

```
MODEL
   PI_MLights
SETS
   ML_Color = {Blue, Green, Red, Yellow} ;
   PI_MLights_STATES = {ML_Off, ML_On}
VARIABLES
   color, ml_state
INVARIANT
   color ∈ ML_Color ∧
   ml_state ∈ PI_MLights_STATES
INITIALISATION
   color, ml_state := ML_Blue, ML_Off
OPERATIONS
   change(new) =
      PRE new ≠color ∧ ml_state = ML_Off
      THEN color := new
      END ;
   on =
      PRE ml_state = ML_Off
      THEN ml_state := ML_On
      END ;
   off =
      PRE ml_state = ML_On
      THEN ml_state := ML_Off
      END
END
```

(b) PI_MLights

```
MODEL
   PI_SLight
SETS
   PI_SLight_STATES = {Stop, Start}
VARIABLES
   sl_state
INVARIANT
   sl_state ∈ PI_SLight_STATES
INITIALISATION
   sl_state := Stop
OPERATIONS
   start =
      PRE sl_state = Stop
      THEN sl_state := Start
      END ;
   stop =
      PRE sl_state = Start
      THEN sl_state := Stop
      END
END
```

(c) PI_SLight

Figure 5: B Models of the interfaces of the three components

3 Definition and Verification of Adapters

We must now prove that the controller can be connected either with the MultiLights component or with two versions of the SingleLight component. A process of proving interoperability between components using the B refinement is described in [7]. We can show that the interface PI_MLights of the MultiLights component is not a B refinement of the RI_Lights interface of the Controller component, because we have no direct matchings. The refinement proof fails, showing that the two components cannot be directly connected.

As specified in Figure 1, we introduce an adapter, i.e. a piece of code that takes place between the both considered components and compensates for the difference between their interfaces. Intuitively, this adapter proposes a way to connect provided operations and attributes to the required ones. Let us consider two components, one with a requested interface RI and the other with a provided interface PI. We define an adapter between these two components as a *new* component that *realizes* or implements the required interface RI, *using* the provided interface PI.

In order to verify the interoperability between RI and PI, we propose to specify the adapter as a B model. As shown in Figure 6, the B model Adapter_1

1. **REFINES** the B model of the required interface : the adapter is an "implementation" of the required interface and
2. **INCLUDES** the B model of the provided interface. The adapter uses "correctly" the operations of the provided interface to implement the required interface.

The B specification of this adapter is composed of two main parts:

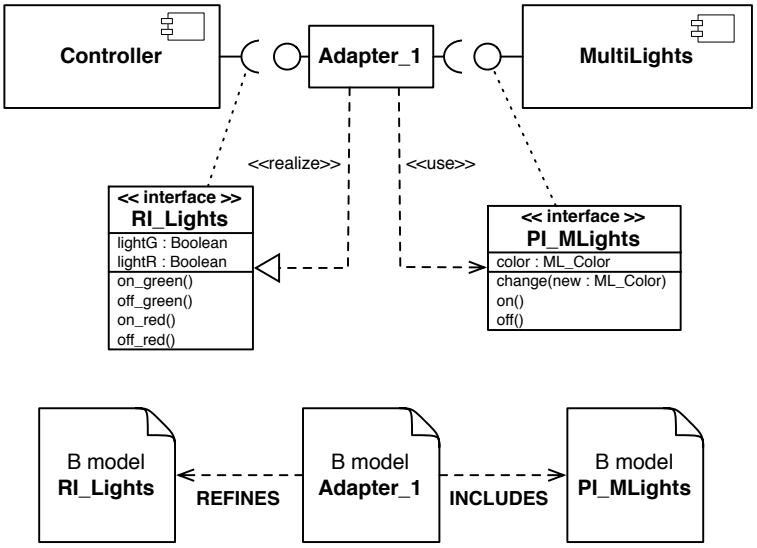

Figure 6: An Adapter between Controller and MultiMLights components

- The **INVARIANT** clause describes the links between the attributes of both required and provided interfaces (linking invariant). For each required variable, the adapter must specify how to obtain it in terms of the provided variables. The adapter must also express the links between the control states of both corresponding PSMs.
- The **OPERATIONS** clause is composed of all the operations of the required interface. The body of each operation is defined by the call of some operations of the provided interface linked together by a small part of code.

The use of the B method and its refinement mechanism allows us to verify that the proposed adapter is a "correct" refinement of the B model of the required interface RI, at the three levels of interoperability:

- the syntactic level is verified by the linking invariant concerning the attributes and by the correspondence of operations between the required interface and the adapter model,
- the semantic level is checked in terms of the B refinement: for each operation of the B adapter model, its precondition must imply, under its invariant, the precondition of the corresponding operation of the required model; the application of its substitutions must preserve the linking invariant,
- the protocol constraints expressed by PSMs have been transformed into preconditions and substitutions of the B operations. The protocol level is taken into account by behavioral constraints during the proof of the refinement.

It is to be noticed that behavioral and protocol constraints expressed in the provided interface and translated into the corresponding B specification are also taken into account. When an operation of the provided interface specification is used into the adapter specification (this is possible by the use of the B **includes** clause), its precondition is verified.

3.1 An Adapter for the MultiLights Component

As presented Figure 6, this adapter uses the provided interface PI_MLights implemented by the MultiLights component in order to realize the required interface RI_Lights of the Controller component. The B specification of Adapter_1 is given Figure 7:

- its invariant expresses the required interface attributes $lightG$ and $lightR$ in terms of the provided attributes $color$ and ml_state. It is to be noticed that lr_state does not correspond to an interface attribute. As previously explained, it has been introduced to express the control states,
- the **OPERATIONS** clause expresses how each required operation is implemented in terms of the provided operations. For example, the operation $on\_green()$ is defined by the choice of the suitable color (if necessary) before turning on the light; the sequential operation calls is expressed in B by a ";" statement and the choice, by an "if... then... else..." statement.

```
REFINEMENT
   Adapter_1
REFINES RI_Lights
INCLUDES PI_MLights
INVARIANT
   lightG = bool(color = Green ∧ ml_state = ML_On) ∧
   lightR = bool(color = Red ∧ ml_state = ML_On) ∧
   (lightG = FALSE ∧ lightR = FALSE ⇒ ml_state = ML_Off)
OPERATIONS
   on_green =
      IF color = Green
      THEN on
      ELSE
         LET col BE col = Green
         IN change(col) ; on
         END
      END ;
   off_green =
      BEGIN off
      END ;
   on_red =
      IF color = Red
      THEN on
      ELSE
         LET col BE col = Red
         IN change(col) ; on
         END
      END ;
   off_red =
      BEGIN off
      END
END
```

Figure 7: B Model of Adapter_1 between Controller and MultiLights Components

3.2 An Adapter for the SingleLight Component

Using the SingleLight component to realize the required interface RI_Lights of the Controller component is not immediate. The needed functionalities implies two colors for the lights, even if they are not on at the same time. As the SingleLight component provides only one light of one color, the adapter will use two instances of this component, namely PI_Green::PI_SLight and PI_Red::PI_SLight, to answer the required needs. The schema of the architecture is presented Figure 8. It is the same as the general schema presented at the beginning of Section 3, with the use of two provided interfaces.

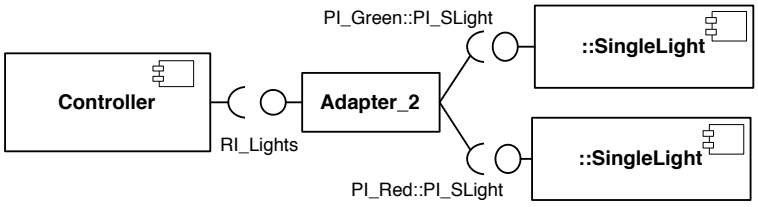

Figure 8: Architecture of the system when using the SingleLight component

The B specification of Adapter_2 is given Figure 9: it includes two instances PI_Green and PI_Red of the B model of PI_SLight. Once we have made the choice of using two different instances of this component, the definition of the link is immediate. The required attributes $lightG$ and $lightR$ and the operations are directly expressed in terms of the operations of the two instances of PI_SLight.

```
REFINEMENT
   Adapter_2
REFINES RI_Lights
INCLUDES PI_Green.PI_SLight, PI_Red.PI_SLight
INVARIANT
lightG = bool(PI_Green.sl_state = Start) ∧
lightR = bool(PI_Red.sl_state = Start)
OPERATIONS
   on_green =
      BEGIN PI_Green.start
      END ;
   off_green =
      BEGIN PI_Green.stop
      END ;
   on_red =
      BEGIN PI_Red.start
      END ;
   off_red =
      BEGIN PI_Red.stop
      END
END
```

Figure 9: B Model of Adapter_2 between Controller and two instances of SingleLight

3.3 Verification of the interoperability

We use the B4free tool [2] to verify that Adapter_1 and Adapter_2 refine the required interface RI_Lights. The verification results are as follows:

- B4free generates 14 obvious proof obligations for the B model of Adapter_1. All these proof obligations were proven automatically,
- B4free generates 4 obvious proof obligations for the B model of Adapter_2. All these proof obligations were proven automatically.

According to these results, we conclude that Adapter_1 **refines** RI_Lights using the MultiLights component and Adapter_2 **refines** RI_Lights using two instances of the SingleLight component. Consequently, each adapter we have considered implements the requested interface in terms of services provided by the corresponding component. The interoperability is verified at the signature, semantic and protocol levels.

4 Related Work

The main drawback of component-based software engineering is the high cost of components deployment. This cost comes from the verification of the components interoperability and from the necessary definition of adapters.

In [28, 29], Zaremenski and Wing propose an interesting approach to compare two software components. It determines whether one required component can be substituted for another. They use formal specifications to model the behavior of components and exploit the Larch prover to verify the specification matching of components.

In [6] a subset of the polyadic π-calculus is used to deal with the components interoperability, only at the protocol level. π-calculus is very well suited language for describing component interactions. The main limitation of this approach is the low-level description of the used language and its minimalistic semantic. In [11], 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.

Several proposals for component adaptation have already been made.

Some practice-oriented studies have been devoted to analyze different issues when one is faced to the adaptation of a third-party component [10]. A formal foundation to the notions of interoperability and component adaptation was set up in [27]. Component behaviors specifications are given by finite state machines which are well known and supports simple and efficient verification techniques for the protocol compatibility.

Braccalia and al [4, 5] specify an adapter as a set of correspondences between methods (and parameters) of the both required and provided components. The adapter is formalized as a set of properties expressed in π-calculus. From this specification and from the both interfaces, they generate a concrete (implementable) adapter.

Reussner and Schmit present adapters in the context of concurrent systems. They consider only a certain class of protocol interoperability problems and they generate adapters for bridging component protocol incompatibilities, using interface described by finite parameterized state machine [19, 21, 20].

Our proposition takes benefits from object oriented notations : components are described using high-level UML 2.0 interfaces and their protocol state machines ; it takes also benefits from formal methods and their existing tools support, using existing derivation rules from UML diagrams to B specifications : we propose an adapter as a B specification and the interoperability verification is supported at the signature, semantic and protocol levels in the same framework using the B refinement.

5 Conclusion

To construct a working system out of components, adapters have to be defined. These adapters implement a required interface in terms of some provided interfaces. We have proposed a model of adapters expressed in the B formal method allowing to define rigorously the interoperability between components and to check it with tool support: an adapter is a correct refinement of the B model of the required interface using existing provided components. The interoperability is verified at the signature, semantic and protocol levels.

We want also to take into account more complex adapters. Generally, an adapter may use some provided interfaces offered by different components to realize some other required interfaces by others components. We are currently exploring different kinds of adapters in terms of specification matching [29]. We are working on alternative versions of compatibility and their mappings to refinement in B, in order to give patterns for the corresponding adapters in the same framework.

References

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

[2] J.-R. Abrial and D. Cansell. Click'n'Prove : Interactive Proofs Within Set Theory. In D. Basin and B. Wolff, editors, *16th International Conference on Theorem Proving in Higher Order Logics - TPHOLs'2003*, volume 2758 of *LNCS*, pages 1–24. Springer Verlag, 2003.

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

[4] A. Braccalia, A. Brogi, and F. Turini. Coordinating Interaction Patterns. In *Symposium on Applied Computing (SAC'2001)*, 2001.

[5] A. Bracciali, A. Brogi, and C. Canal. A formal approach to component adaptation. In *Journal of Systems and Software*, 2005.

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

[7] S. Chouali, M. Heisel, and J. Souquières. Proving Component Interoperability with B Refinement. In H. R. Arabnia and H. Reza, editors, *International Worshop on Formal Aspects on Component Software*, pages 915–920. CSREA Press, 2005. To appear in ENCTS 2006.

[8] 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*, 2005.

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

[10] D. garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch: Why Reuse is so Hard. *IEEE Software*, 12(6):17–26, 1995.

[11] 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.

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

[13] D. Konstantas. Interoperation of object oriented application. In *In O. Nierstrasz and D. Tsichritzis, editors, Object-Oriented Software Composition*, pages 69–95. Prentice Hall, 1995.

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

[15] E. Meyer and J. Souquières. A systematic approach to transform OMT diagrams to a B specification. In *Proceedings of the Formal Method Conference*, LNCS 1708, pages 875—895. Springer-Verlag, 1999.

[16] Microsoft Corporation. *The Component Object Model Specification, Version 0.9*, 1995. http://www.microsoft.com/com/resources/comdocs.asp.

[17] Object Management Group. UML superstructure specification, v2.0, 2005.

[18] The Object Mangagement Group (OMG). *The Common Object Request Broker: Architecture and Specification, Revision 2.2*, February 1998. http://cgi.omg.org/library/corbaiiop.html.

[19] R. H. Reussner. Adapting Components and Predicting Architectural Properties with Parameterised Contracts. In Wolfgang Goerigk, editor, *Tagungsband des Arbeitstreffens der GI Fachgruppen 2.1.4 und 2.1.9, Bad Honnef*, pages 33–43, May 2001.

[20] 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*. 2003.

[21] H. W. Schmidt and R. H. Reussner. Generating adapters fo concurrent component protocol synchronisation. In Ivica Crnkovic, Stig Larsson, and Judith Stafford, editors, *Proceeding of the Fifth IFIP International conference on Formal Methods for Open Object-based Distributed Systems*, 2002.

[22] Steria. *Obligations de preuve: Manuel de référence, version 3.0*.

[23] Sun Microsystems. *JavaBeans Specification, Version 1.01*, 1997. http://java.sun.com/products/javabeans/docs/ spec.html.

[24] Sun Microsystems. *Enterprise JavaBeans Specification, Version 2.0*, 2001. http://java.sun.com/products/ejb/ docs.html.

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

[26] P. Wegner. Interoperability. *ACM Computing Survey*, 28(1):285–287, 1996.

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

[28] A. M. Zaremski and J. M. Wing. Signature matching: a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2):146–170, 1995.

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