An Approach to Specify Components for Interoperability

Maritta Heisel[1] and Jeanine Souquières[2]

[1] Universität Duisburg-Essen
Institut für Medientechnik und Software Engineering
D-47048 Duisburg, Email: maritta.heisel@uni-duisburg-essen.de

[2] LORIA – Université Nancy 2, Campus Scientifique BP 239
F-54506 Vandœuvre lès Nancy cedex, Email: Jeanine.Souquieres@loria.fr

Abstract. We propose an approach for component based development in such a way that interoperability between the different components is proved. It is based on the use of existing notations and languages with their associated tools allowing validation and verification steps: a context diagram for analysing and structuring the problem, a component architecture diagram for the description of the whole system in terms of components and interfaces, sequence diagrams to describe the behaviour of each component and the formal method B for specifying interfaces of the different components.

Keywords: Component based approach, interoperability, formal method

1 Introduction

In recent years, the paradigm of component orientation [13, 23] has become more and more important in software engineering. Its underlying idea is to develop software systems not from scratch but by assembling pre-fabricated parts, as is common in other engineering disciplines. Component orientation has emerged from object orientation, but the units of deployment are usually more complex than simple objects. As in object orientation, components are encapsulated, and their services are accessible only via interfaces and their operations.
In order to really exploit the idea of component orientation, it must be possible to acquire components developed by third parties and assemble them in such a way that the desired behaviour of the software system to be implemented is achieved. This approach leads to the following requirements:

– The *specification* of a component must contain sufficient information to decide whether or not to acquire it for integration in a new software system. This requirement concerns the access to the component's source code that may not be granted in order to protect the component producer's interests. Moreover, component consumers should not be obliged to read the source code of a component to decide if it is useful for their purposes or not. Hence, the source code should not be considered to belong to the component specification.

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

In order to fulfill the above requirements, we propose an approach for component based development, using existing languages and notations with their associated tools to help in the validation and the verification steps. It is based on the use of:

- a context diagrams [15] for analysing and structuring the problem in terms of domains and interfaces,
- UML2.0 diagrams [21] like a component architecture diagram to express the overall architecture of the system, sequence diagrams to express the visible behaviour of the different components involved,
- the B formal method [1] for its underlying concepts of machine and refinement which fit well with components and their interoperability, and because the method is equipped with powerful tool support. Thus, we can exploit existing technology for proving component interoperability. Using for example the object constraint language OCL [26] and generating verification conditions from scratch would be much more tedious.

The rest of the paper is organised as follows. In Section 2 we present an overview of the B method. We introduce our approach in Section 3 and illustrate it on a case study of the access control system 4. In Section 5, we discuss related work. The paper finishes with some concluding remarks in Section 6.

2 The B formal method

The B method [1] is a formal software development approach based on the set theory, allowing to develop software for critical systems. The B method enables an incremental development process, known as a refinement process. A system development begins by the definition of an abstract view which can be refined step by step until an implementation is reached. The refinement over models is a key feature for developing incrementally models from a textually-defined system, while preserving correctness. It implements the proof-based development paradigm [18, 22].

The method has been successfully used in the development of several complex real-life applications, like 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].

Specifications are composed of abstract machines which are very closed to notions well-known in programming under the name of modules, classes or abstract data types. Each abstract machine consists of a set of variables, invariant properties of those variables and operations. The state of the system, i.e. the set of variable values, is modifiable by operations which must preserve its invariant. 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 [20, 17]. Proofs for invariance and refinement are parts of each development. The proof obligations are generated automatically by support tools like AtelierB [22], B-Toolkit [18] and B4free [11], an academic version of AtelierB. The check of proof obligations with B support tools either through automatic or interactive proofs [2], is an efficient and practical way to detect errors introduced during the specification development.

3 A general approach for component based software development

Our goal is to propose an approach based for component based development in such a way that interoperability between the different used components is proved. 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. The approach is decomposed in several steps described as follow:

1. We first set up a generalized context diagram as proposed by Jackson [15]. This diagram is a good guide for the introduction of the different needed interfaces starting from the control of the shared phenomena and the machine domains.
2. We can then set up an architecture of the whole system with provided and required interfaces. We use UML2.0 component architecture diagrams [21]. In this step, components with their required and provided interfaces are identified.
3. For each component of the architecture, we propose to set up a specification containing:
   - Sequence diagrams describing the visible behavior of the specified component; the sequence diagrams may contain all components the specified component is connected with.
   - A B machine for each provided and each required interface. For all interfaces that connect the specified component with the same outside component, the interface data models must be the same, and the IDM must be encoded in the B machine.

The specification of the interface data model specifies (i) the types used in the interface, (ii) a data state as far as necessary to express the effects of operations, and (iii) invariants on that data state. Each machine contains the operations belonging to its corresponding interface. An operation specification consists of its signature (i.e., the types of its input and output parameters), its precondition expressing under which circumstances the operation may be invoked, and its postcondition expressing the effect of the operation. Both pre- and postcondition will refer to the interface data model.

4. When constructing a software system from existing components, the components must be connected in an appropriate way. To achieve this, for each connection of a provided and a required interface contained in the architecture:
   - we show that, may be after some syntactic transformations, the provided interface is a B refinement of the required interface [9];
   - if this is not possible, we either try to change the specification of a machine component, or we try to develop an adapter;
   - if the latter approach is not possible either, the components cannot be connected as shown in the architecture.

Each B specification contains a state invariant, allowing to make a link with sequences diagrams expressing the behaviour of the component, dependencies and an usage protocol, i.e. the order in which the operations may be invoked. It also contains an initial state.

When a component manipulates data, it is possible to use a UML class diagram to express the data model for reasons of readability. This class diagram is then automatically transformed into a B specification [20].

4 Case study: a simplified access control system

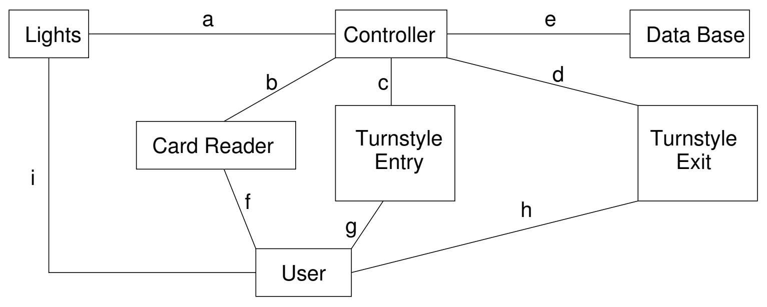
The idea is to elaborate a system which will be able to control the access of certains persons to a given building. The control takes place on the basis of the authorization that each person who is concerned is supposed to possess. When somehone is inside the building, his eventual exit must also be controlled by the system, so as to be able to know at any moment how many persons are inside the building. The entry or the exit of the building follows a systematic procedure. We summerize the access protocol as follow:

- Each user involved received a magnetic card with a unique identifying sign, which is engraved on the card reader. To enter the building, the user has to insert the card into a card reader installed at the entrance of the building. There is a data base where the information who is authorized to enter the building is stored. According to that information, access is granted or not.

- If access is granted, a green light near the card reader is turned on for some time, the card is ejected, and the entry turnstile which is normally blocked is unblocked. Nobody can get through this turnstile without being controlled by the system. This turnstile is only affectd to a single task, the entry.
- The entry turnstile is re-blocked either after entry of the user or after some timeout. If the user does not take the card in some time limit, the card is retracted and kept.
- If access is denied, a red light is turned on for some time, and the entry turnstile remains blocked.
- The number of persons present in the building must be counted. Therefore, there is also an exit turnstile which is never blocked, but just serves to observe when a person leaves the building.

4.1 Context diagram

A context diagram for this simplified access control system is proposed Figure 1. We can see that the user interact with the different components except with the data base one which is a passive component.



a: C!{green_on, green_off, red_on, red_off}
b: C!{eject_card_c, retract_card_c}
CR!{card_inserted, card_taken}
c: C!{lock_c, unlock_c}
TE!{entered}
d: TX!{left}
e: C!{enter_i, leave_i}
DB!{authorized}

f: CR!{eject_card, retract_card}
U! {insert_card, take_card}
g: U!{entry} TE! {block, unblock}
h: U!{leave}
i: L!{see_green, see_red}

Fig. 1. Context diagram for the access control system

4.2 Architecture of the system

The architecture of the system is depicted Figure 2, using the component architecture diagram of UML2.0. We have used naming conventions for interfaces. Each interface name has the form XYZ, where

– X is the abbreviation of the name of the specified component, i.e. the first letter;
– Y is either "P" for provided or "R" for required;
– Z is the abbreviation of the name of the component the specified component is connected to.

As an example, LPC is the provided interface of the Lights component which is connected to the Controller component.

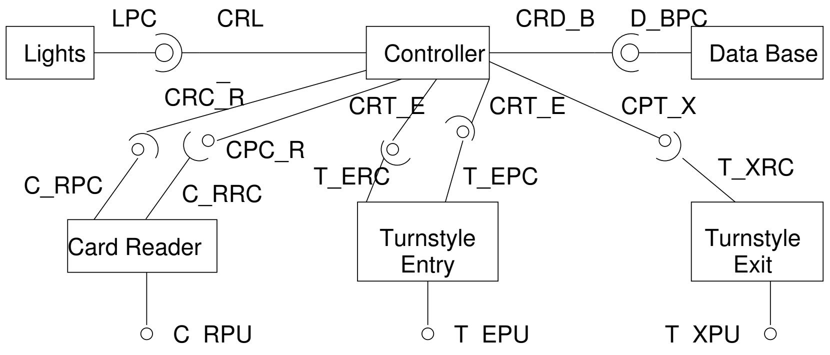


Fig. 2. Architecture of the global access control system

Note that even though the phenomenon *authorized* is controlled by the data base, this does not correspond to a required interface of the data base connected to the controller. In fact, the data base is a passive component without any influence on the behaviour of the system. Hence it only has a provided interface.

4.3 The card Reader component

Let us specify one of the required component for the access control system, the Card Reader one. Its three interfaces are:

– C_RPC, its provided interface relatively to the Controller component. The two operations $eject_card_c$ and $retract_card_c$ are controlled by the Controller;
– C_RRC, its request interface relatively to the Controller component. The two operations $card_inserted$ and $card_taken$ are controlled by the Card Reader;
– C_RPU, its provided interface relatively to the User component. The two operations $insert_card$ and $take_card$ are controlled by the User.

Specification of the behaviour of the Card Reader component

To describe the behaviour of this component, a sequence diagram is introduced Figure 3. Two other objects are used, namely the the controller and the reader corresponding to the two components connected to the card reader.

Maritta: je propose ici de mettre les trois diagrammes en un seul, celui du comportement global du composant. C'est une concaténation des 3. Je ne comprends pas pourquoi tu l'as décomposé.

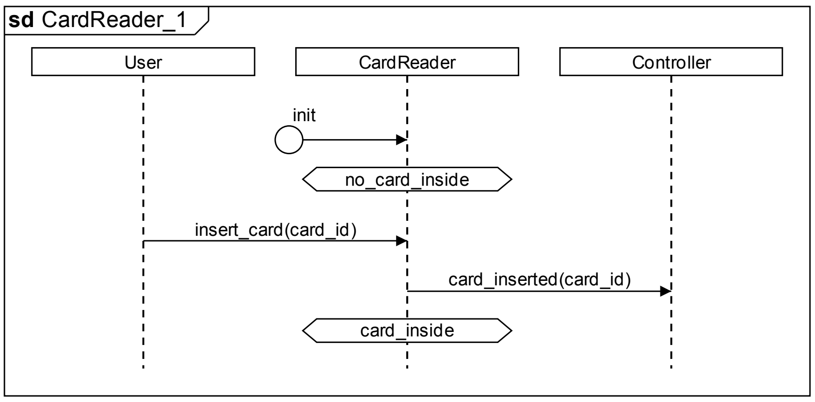


Fig. 3. Sequence diagram for the Card Reader component

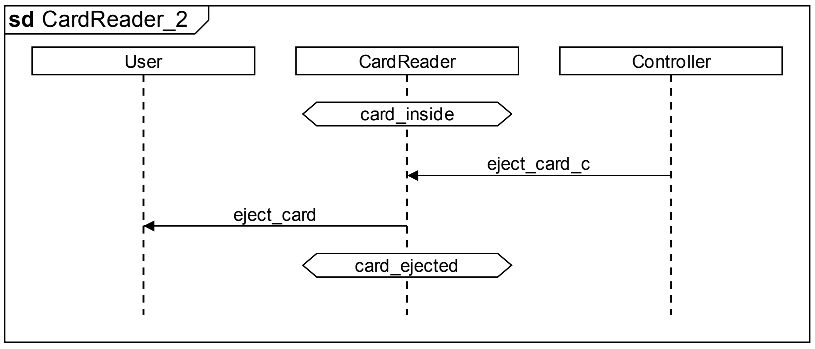


Fig. 4. Sequence diagram for the Card Reader component

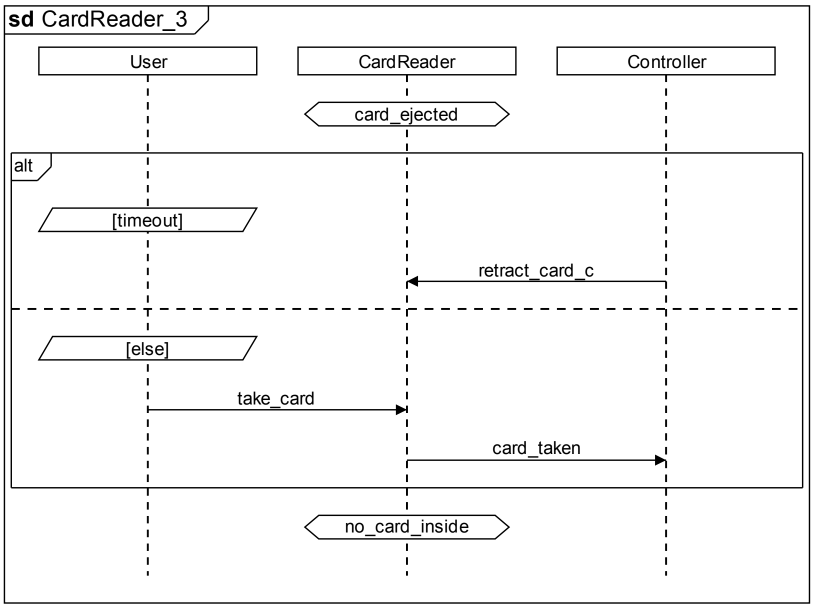


Fig. 5. Sequence diagram for the Card Reader component

Specification of the interface C_RPC

```
MACHINE  C_RPC
  SETS  CardReaderStates = {card_inside, card_ejected, no_card_inside};      CardID
  CONSTANTS  no_card
  PROPERTIES  no_card ∈ CardID
  VARIABLES  state, nb_cards_kept, current_card
  INVARIANT  state ∈ CardReaderStates ∧
          nb_cards_kept ∈ ℕ ∧ current_card ∈ CardID ∧
          state = card_inside ⇔ current_card ≠ no_card
  INITIALISATION  state := state := no_card_inside ||
          nb_cards_kept := 0 ||
          current_card := no_card
  OPERATIONS
    eject_card_c ==
          PRE  state = card_inside
          THEN  state := card_ejected ||
              current_card := no_card
          END
    retract_card_c ==
          PRE  state = card_ejected
          THEN  state := no_card_inside
              || nb_cards_kept := nb_cards_kept+1
          END
END
```

Specification of the interface C_RPU

```
MACHINE C_RPU
  SETS CardReaderStates = {card_inside, card_ejected, no_card_inside};
         CardID
  CONSTANTS no_card
  PROPERTIES $no\_card \in CardID$
  VARIABLES state, nb_cards_kept, current_card
  INVARIANT $state \in CardReaderStates \wedge$
         $nb\_cards\_kept \in \mathbb{N} \wedge current\_card \in CardID \wedge$
         $state = card\_inside \Leftrightarrow current\_card \neq no\_card$
  INITIALISATION state := state := no_card_inside ||
         nb_cards_kept := 0 ||
         current_card := no_card
  OPERATIONS
    insert_card(card_id) ==
         PRE $state = no\_card\_inside \wedge card\_id \in CardID$
         THEN state := card_inside || current_card := card_id
         END
    take_card ==
         PRE state = card_ejected
         THEN state := no_card_inside
         END
END
```

Specification of the interface C_RRC

```
MACHINE C_RRC
...
  OPERATIONS
    card_inserted(card_id) ==
         PRE $state = card\_inside \wedge card\_id \in CardID$
         THEN skip
         END
    card_taken ==
         PRE state = card_ejected
         THEN state := no_card_inside
         END
END
```

4.4 Specifying the controller component

In the same way, we have to specify the Controller component. Since the controller communicates with all other components, we get one quite complex sequence diagram for the entry protocol, and a simple one for the exit protocol. Its interfaces are:

– CRL, its request interface relatively to the Light component. The for operations $green_on$, $green_off$, red_on and red_off are controlled by the Controller;
– $C_C RC_R$, its request interface and CPC_R its provided interface relatively to the Card Reader component.
– ...

We have to verify that the global behaviour is satisfied by the behaviour of each component.

Verification for the Card Reader component Let us see the connexion with the Card Reader component.

Sequence diagram for the entrance of a user

Donner ici le diagramme (cf. note aout 2005, page 8 avec un nuage). We note that all the operations provided by the Card Reader are used.

– Here, we use synchronous messages for the first time:
  The message *card_inserted*(*card_id*) sent from the card reader is followed by a call to the data base *authorized*(*card_id*), which has a boolean result.
– What is the specification of the operation *card_inserted*(*card_id*)? Intuitively, I would say, calling the operation *authorized*(*card_id*). But how could we express this in B? The different operations belong to different interfaces!
– How should we specify the IDM of the interface CPC_R? It must be at least as rich as the IDM of C_RRC! Otherwise, CPC_R could not be a refinement of C_RRC.

Parler du raffinement

Verification for the Turnstile component It is used twice, once for the entry, once for the exit.
Problem of initial state.

5 Related work

In an earlier paper, we have investigated the role of component models in component specification [14]. The specification of a component model makes it possible to obtain more concise specifications of individual components, because these may refer to the specification of the component model. The component model specification need not be repeated for each individual component adhering to the component model in question. In this paper, we investigate the necessary ingredients a component specification must have in order to be useful for assembly of a software system out of components. These ingredients are independent of concrete component models. Several proposals for component specification have already been made. They have in common that they have no counterpart of our

interface data model and that they do not consider interoperability issues, but only the specification of single components.
A working group of the German "Gesellschaft für Informatik" (GI) has defined a specification structure for business components [24]. That structure comprises seven levels, namely marketing, task, terminology, quality, coordination, behavioral, and interface. Our specification structure covers the layers terminology, coordination, behavioral, and interface by proposing concrete ways of specifying each of those levels. The other layers of the GI proposal have to do with non-functional aspects of components.
Beugnard et al. [6] propose to define contracts for components. They distinguish four levels of contracts: syntactic, behavioral, synchronization, and quality of service. The syntactic level specifies only the operation signatures, the behavioral level contains pre- and postconditions, the synchronization level corresponds to usage protocols, and the quality of service level deals with non-functional aspects. Beugnard et al. do not introduce data models for their interfaces. It cannot easily be checked if two components can be combined.
The component specification approach of Lau and Ornaghi [16] is closer to ours, because there, each component has a *context* that corresponds to our interface data model. A context is an algebraic specification, consisting of a signature, axioms, and constraints. In contrast, we deem it more appropriate to allow for an object-oriented specification of the data model of a component interface. This makes it possible to take side effects of operations into account and to use inheritance, concepts that are frequently used in practice.
Cheesman and Daniels [8] propose a process to specify component-based software. This process starts with an informal requirements description and produces an architecture showing the components to be developed or reused, their interfaces and their dependencies. For each interface operation, a specification is developed, consisting of a precondition, a postcondition and possibly an invariant. This approach follows the principle of design by contract [19]. Our specification of component interfaces is inspired by Cheesman and Daniels' work because that work clearly shows that for each interface, a data model is necessary. However, Cheesman and Daniels do not consider the case that already existing components with possibly different data models have to be combined, and hence they do not define a notion of interoperability.
Canal et al. [7] use a subset of the polyadic π-calculus to deal with component interoperability only at the protocol level. The π-calculus is well suited for describing component interactions. The limitation of this approach is the low-level description of the used language and its minimalistic semantics.
Bastide et al. [3] use Petri nets to specify the behavior of CORBA objects, including operation semantics and protocols. The difference with our approach is that we take into account the invariants of the interface specifications.
Zaremski and Wing [27] propose an interesting approach to compare two software components. It is determined whether one component can be substituted for another. They use formal specifications to model the behavior of components and the Larch prover to prove the specification matching of components.

Others [12, 25] have also proposed to enrich component interface specifications by providing information at signature, semantic and protocol levels. Despite these enhancements, we believe that in addition, a data model is necessary to perform a formal verification of interface compatibility.
The idea to define component interfaces using B has been introduced in an earlier paper [10]. The use of the B refinement to prove that two components are compatible at the signature and semantics levels has been explored in [9].

6 Conclusion

We have presented a manner of specifying component interfaces that is independent of specific component models.

References

1. J.-R. Abrial. *The B Book.* Cambridge University Press - ISBN 0521-496195, 1996.
2. J-R. Abrial and D. Cansell. Click'n'Prove : Interactive Proofs Within Set Theory. In D. Basin et B. Wolff, editor, *16th International Conference on Theorem Proving in Higher Order Logics - TPHOLs'2003*, volume 2758 of *LNCS*, pages 1–24. Springer Verlag, 2003.
3. R. Bastide, O. Sy, and P. A. Palanque. Formal specification and prototyping of CORBA systems. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 474–494. Springer-Verlag, 1999.
4. P. Behm, P. Benoit, and J.M. 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. A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *IEEE Computer*, pages 38–45, July 1999.
7. C. Canal, L. Fuentes, E. Pimentel, J-M. Troya, and A. Vallecillo. Extending CORBA interfaces with protocols. *Comput. J.*, 44(5):448–462, 2001.
8. J. Cheesman and J. Daniels. *UML Components – A Simple Process for Specifying Component-Based Software.* Addison-Wesley, 2001.
9. S. Chouali, M. Heisel, and J. Souquières. Proving Component Interoperability with B Refinement. In ENCTS, editor, *FACS'05, International Worshop on Formal Aspects on Component Software*, 2005.
10. S. Chouali and J. Souquières. Verifying the compatibility of component interfaces using the B formal method. In CSREA Press, editor, *SERP'05, International Conference on Software Engineering Research and Practice*, To appear, June 2005.
11. Clearsy. B4free. Available at *http://www.b4free.com*, 2004.
12. 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.
13. G. T. Heineman and W. T. Councill. *Component-Based Software Engineering.* Addison-Wesley, 2001.
14. M. Heisel, T. Santen, and J. Souquières. Toward a formal model of software components. In Chris George and Miao Huaikou, editors, *Proc. 4th International Conference on Formal Engineering Methods*, LNCS 2495, pages 57–68. Springer-Verlag, 2002.

15. M. Jackson. *Problem Frames. Analyzing and structuring software development problems.* Addison-Wesley, 2001.
16. K.-K. Lau and M. Ornaghi. A formal approach to software component specification. In G.T. Leavens D. Giannakopoulou and M. Sitaraman, editors, *Proceedings of Specification and Verification of Component-based Systems Workshop at OOPSLA2001*, pages 88–96, 2001.
17. H. Ledang and J. Souquières. Modeling class operations in B: application to UML behavioral diagrams. *ASE2001: 16th IEEE International Conference on Automated Software Engineering, IEEE Computer Society*, 2001.
18. B-Core(UK) Ltd. *B-Toolkit User's Manual.* Oxford (UK), 1996. Release 3.2.
19. B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, second edition, 1997.
20. 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.
21. OMG. UML 2.0 Superstructure Specification. Available at http://www.omg.org, 2004.
22. Steria. *Obligations de preuve: Manuel de référence.* Steria - Technologies de l'information, version 3.0. Available at *http://www.atelierb.societe.com.*
23. C. Szyperski. *Component Software.* ACM Press, Addison-Wesley, 1999.
24. K. Turowski, editor. *Standardized Specification of Business Components.* Gesellschaft für Informatik, 2002.
25. A. Vallacillo, J. Hernandez, and M. Troya. Object interoperability. In *Object Oriented Technology: ECOOP'99 Workshop Reader*, pages 1–21, 1999.
26. J. Warmer and An. G. Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley, 1999.
27. A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, 1997.