Enhancing Dependability of Component-based Systems

Denis Hatebur[1], Maritta Heisel[1], Arnaud Lanoix[2] and Jeanine Souquières[2]

[1] Universität Duisburg-Essen, Abteilung Informatik und Angewandte Kognitionswissenschaft, D-47048 Duisburg,
{Denis.Hatebur,Maritta.Heisel}@uni-duisburg-essen.de
[2] LORIA – Université Nancy 2, Campus Scientifique BP 239
F-54506 Vandœuvre lès Nancy cedex,
{Arnaud.Lanoix,Jeanine.Souquieres}@loria.fr

Abstract. We present a method to add dependability features to component-based software systems. The method is applicable if the dependability features add new behavior to the system, but do not change its basic functionality. The idea is to start with a layered software architecture whose central component is an application component that covers the behavior of the system for the normal case. It is then possible to enhance the system by adding dependability features in such a way that the central application component remains untouched. Adding dependability features necessitates to change the overall system architecture by replacing or newly introducing software or hardware components. To the initial software architecture, however, only new adapter components have to be added; the other software components need not be changed. Thus, the dependability of a component-based system can be enhanced in an incremental way. Using the formal method B, we derive specifications for the adapters and show that the components of the enhanced architecture interoperate as intended.

1 Introduction

Component orientation is a new paradigm for the development of software-based systems. The basic idea is to assemble the software by combination of pre-fabricated parts (called software components), instead of the developing it from scratch. This procedure resembles the construction methods applied in other engineering disciplines, such as civil or mechanical engineering.

Software components are put together by connecting their interfaces. A *provided* interface of one component can be connected with a *required* interface of another component if the provided interface offers the services needed to implement the required interface. Sometimes, an *adapter* may be necessary to map the required services to the provided ones.

Hence, an appropriate description of the provided and required interfaces of a software component is crucial for component-based development. In earlier papers [1,2], we have investigated how to formally specify interfaces of software

components and how to demonstrate their interoperability, using the formal method B.

In the present paper, we study how dependability features, such as safety, security or fault tolerance features, can be added to component-based software. The goal is to retain the initial software components as far as possible and only add new software components in a systematic way. This approach works out if the inital software architecture is structured in such a way that the core functionality is clearly separated from auxiliary functionality that is needed to connect the components implementing the core functionality to their environment.

To make a software-based system more dependable, new components (hardware or software) must be added, or existing components must be replaced by more dependable ones, while the core functionality remains the same. As a consequence, new or modified interfaces must be taken into account. In order to connect the existing interfaces of the "core" components to the new or modified ones introduced with the addition of dependability features, adapter components must be developed. These adapters "shield" the core components by intercepting and possibly modifying their inputs and outputs. We show how the specifications of the adapters can be derived from the specifications of the interfaces they must connect.

In Section 2 we describe how we support component-based development using the formal specification language B. We then describe our method to add dependability features in Section 3. The method is illustrated by the case study of an access control system, presented in Section 4. The paper closes with the discussion of related work in Section 5 and concluding remarks in Section 6.

2 Using B for Component-Based Development

We first briefly describe the formal language B and then explain how we use B in to context of component-based software and system development.

2.1 The Formal Method B

The B method [3] is a formal software development method based on set theory. Because of its rigor and powerful tool support, it is often used to develop software for critical systems. The B method supports an incremental development process, using refinement. A development begins with the definition of an abstract specification, which can be refined step by step until an implementation is reached.

The method has been successfully applied in the development of several complex real-life applications, such as the METEOR project [4]. It is one of the few formal methods which has robust and commercially available support tools for the entire development life-cycle from specification down to code generation [5].

B specifications consist of abstract machines, which are very close to notions well-known in programming under the names 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 machine, i.e. the set of variable values, is modifiable by operations, which must preserve its invariant. The invariant clause characterizes the meaningful states that are permitted for the machine. The machine should never arrive at a state in which some part of the invariant clause is false.

The B method provides structuring primitives that allow one to compose machines in various ways. Proofs of invariance and refinement are part of each development. The proof obligations are generated automatically by support tools such as AtelierB [6] or B4free [7], an academic version of AtelierB. Checking proof obligations with B support tools (either through automatic or interactive proofs) [8], is an efficient and practical way to detect errors introduced during development.

2.2 Specifying Component Interfaces with B

We define software as well as hardware components of a component-based system by UML sequence diagrams describing the visible behavior of the specified component, and a B machine for each provided and each required interface specifying:

- the types used in the interface
- a data state as far as necessary to express the effects of operations
- invariants on that data state

Each machine specifies 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.

2.3 Proving Interoperability of Component Interfaces

In component-based development, the components must be connected in an appropriate way. To guarantee interoperability of components, we must consider each connection of a provided and a required interface contained in a system or software architecture and try to show that – after some syntactic transformations – the provided interface is a B refinement of the required interface. This means that the provided interface constitutes an implementation of the required interface, and we can conclude that the two components can be connected as intended. The process of proving interoperability between components is described in [1].

3 Adding Dependability Features to Component-Based Software

4 Case Study

We illustrate our purpose with the case study of a simple access control system which manages the access of authorized persons to existing buildings [**?**]. Persons who are authorized to enter the buildings have to be identified. Turnstiles block the entrance and the exit of each building until an authorization is given whereas identification systems are installed at each entrance and exit of the concerned buildings. The system communcates with a data base which know information about authorized persons and persons present in the buildings.

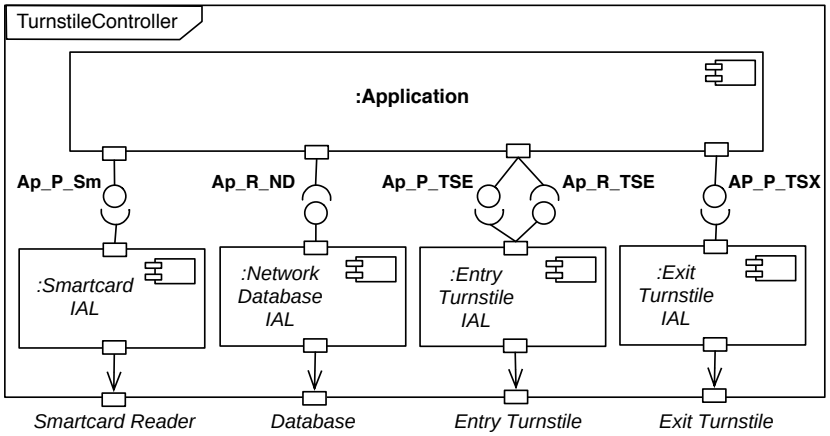


Fig. 1. Required software architecture for TurnstileController

We will concentrate on the software components of the TurnstileController which can be seen as a layered software architecture as presented in the UML 2.0 composite structure diagram of Figure 1. It consists of an Application component and further software components, the drivers, that are needed to communicate with the hardware components. The component Application interacts with four components and its interfaces are described Figure 2. Each interface is modelled by a class diagram with its attributes (if needed) and methods. For example, AP_P_Sm corresponds to its provided interface related to the SmartcardIAL component with one method, namely Card_inserted which is parameterized by a user identifier.

We dispose of three existing COTS components, namely SmartcardIAL, NetworkDatabaseIAL and TurnstileIAL. Each IAL component, i.e. interface abstraction layer, is described Figure 3.

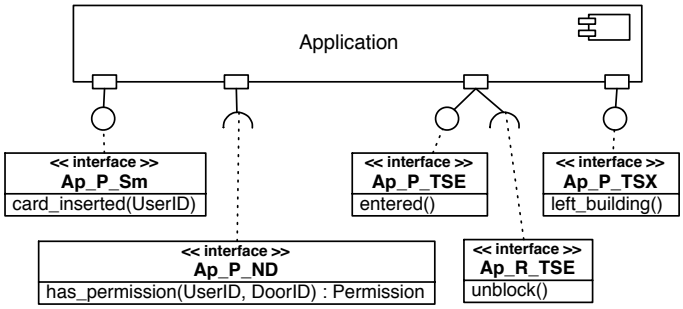
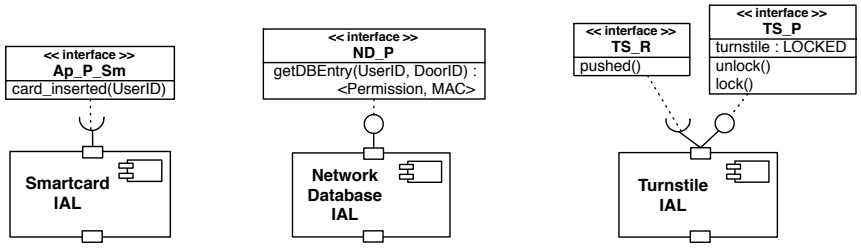


Fig. 2. The different interfaces of Application



(a) SmartcardIAL (b) NetworkDatabaseIAL (c) TurnstileIAL

Fig. 3. COTS components used by TurnstileController

4.1 Description of the System for the Normal Case

When loocking at the different interfaces of the COTS and the Application component, we can see that the SmartcardIAL can be used directly. It is not the case for the data base and the turnstile. in order to correctly connect these two COTS components to realize the required application, adapters have to be introduced as presented Figure 4. It is to be noted that the turnstile component is used twice with a different adapter.

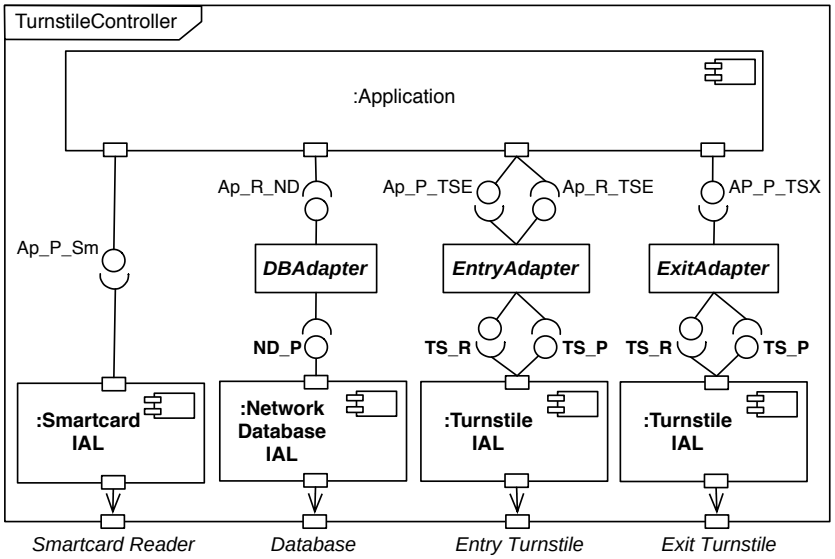


Fig. 4. Software Architecture for TurnstileController

The DBAdapter. The B architecture of DBAdapter is given Figure 5. This schema traduces the adaptation protocol between the required interface Ap_R_ND of the application and the provided interface ND_P of the NetworkDatabaseIAL component, expressed by the UML 2.0 sequence diagram of Figure 6.

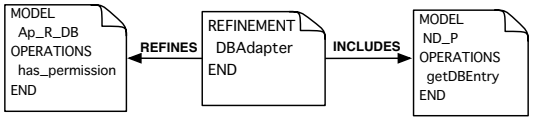


Fig. 5. B architecture for DBAdapter

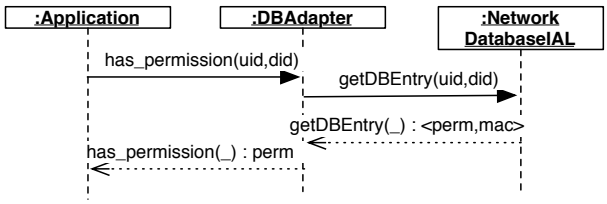


Fig. 6. Sequence diagram for DBAdapter

The EntryAdapter. This adapter is more complex than the previous one because both kinds of interfaces are needed for both components. As presented Figure 7, the sequence diagram shows the sequence of the operation calls between the two interfaces of Application and TurnstimeIAL. This is traduced by the B architecture given Figure 8.

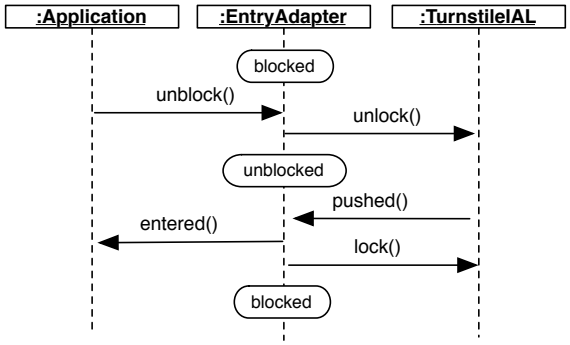


Fig. 7. Sequence diagram for EntryAdapter

4.2 Adding dependability features to the Previous System

Let us now introduce a safety and a security mechanisms. The security mechanism concerns the data base. Its content is now checked using a message authentification code (called mac). A new component called Secret is introduced for storing a secret. The DBAdapter that connects the Application component to the data base is changed to communicate with the Secret component. The Application component stays unchanged.

The safety mechanism concerns the reaction to fire. If a fire occurs, the entry turnstile must remain locked: nobody is allowed to enter the building (we assume the fire brigade uses another entry). Here, the EntryAdapter has to be changed to communicate with the fire detection component.

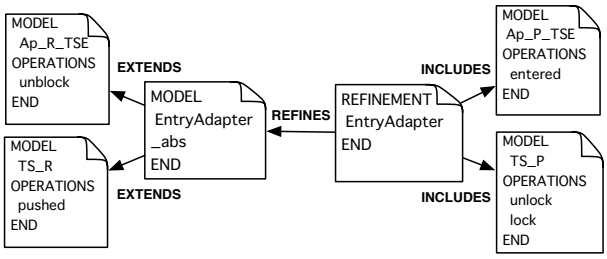


Fig. 8. B architecture for EntryAdapter

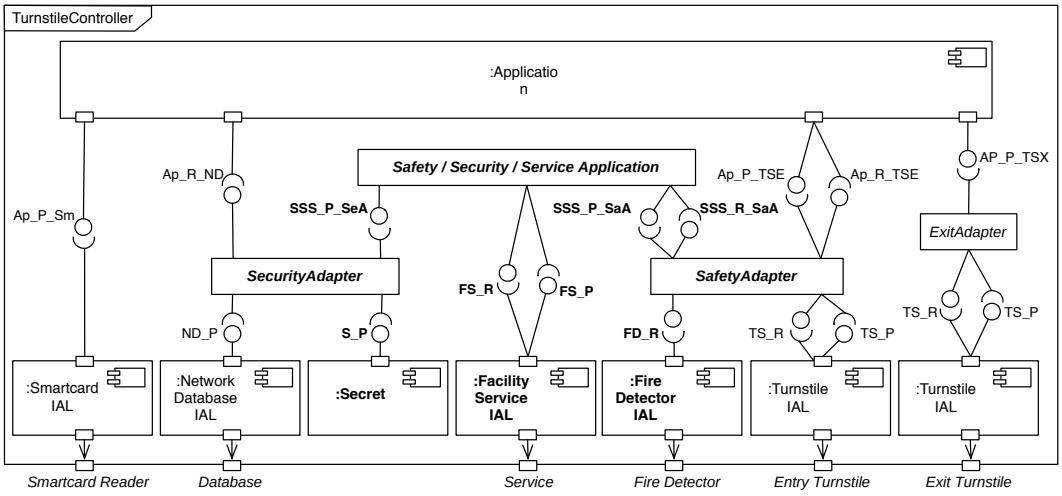


Fig. 9. Safety and Security Software Architecture for TurnstileController

Figure 9 describes the new software architecture of the TurnstileController wchich takes into account these two safety and security policies. Adapters deal now with these policies:

- SecurityAdapter checks the message authentication code and the signature for each database entry and notifies the Safety / Security / Service Application in case of a violation.
- SafetyAdapter blocks the Entry turnstile in case of a fire and informs the Safety / Security / Service Application. The turnstile must be unblocked using this SafetyAdapter.

The SecurityAdapter*.* Figure 10 presents a sequence diagram that introduce security policy between the different involved components. The B architecture of this adapter is given Figure 11.

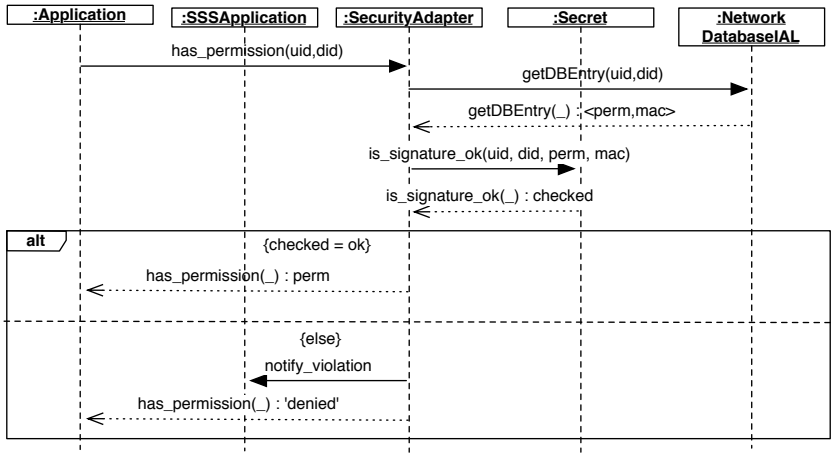


Fig. 10. Sequence diagram for SecurityAdapter

The SafetyAdapter*.* Figure **??** shows a sequence diagram that explains the safety reaction of the adapter when it receives a fire_detected call : the turnstile will be locked until the fire alert will be canceled.

5 Related Work

6 Conclusion and Perspectives

References

1. 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

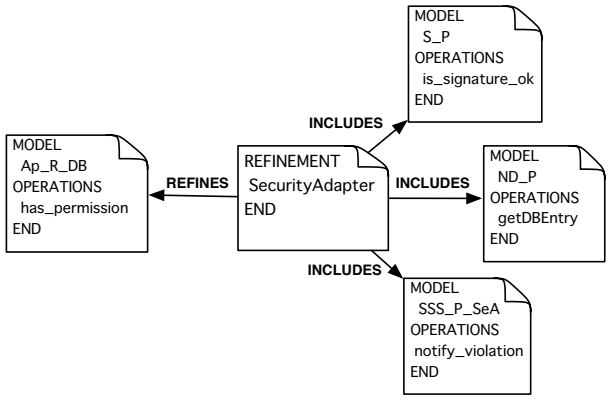


Fig. 11. B architecture for SecurityAdapter

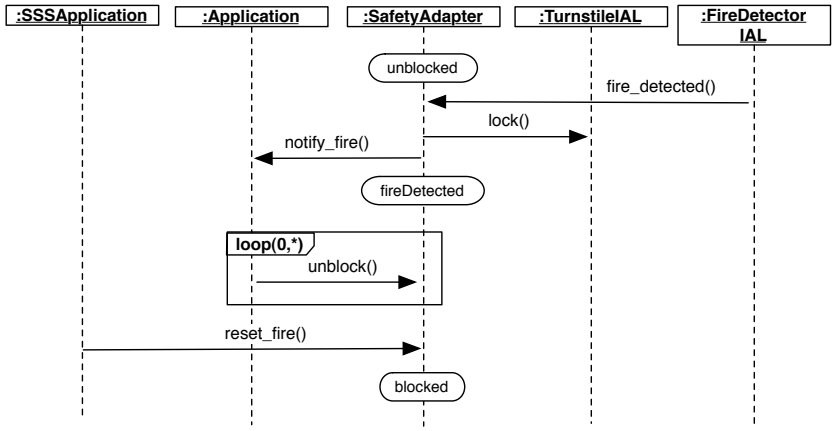


Fig. 12. Sequence diagram for SafetyAdapter

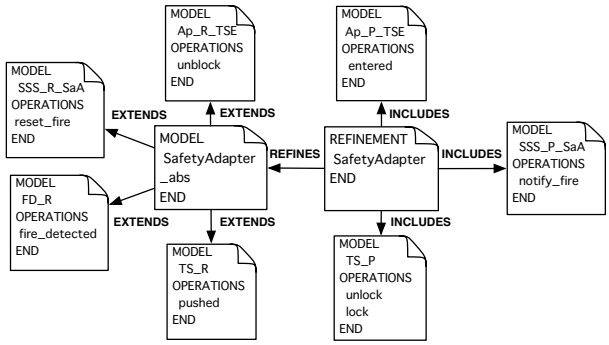


Fig. 13. B architecture for SafetyAdapter

2. Hatebur, D., Heisel, M., Souquières, J.: A method for component-based software and system development. In: Proc. Euromicro. (2006)
3. Abrial, J.R.: The B Book. Cambridge University Press - ISBN 0521-496195 (1996)
4. Behm, P., Benoit, P., Meynadier, J.: METEOR: A Successful Application of B in a Large Project. In: Integrated Formal Methods, IFM99. Volume 1708 of LNCS., Springer Verlag (1999) 369–387
5. Bert, D., Boulmé, S., Potet, M.L., Requet, A., Voisin, L.: Adaptable Translator of B Specifications to Embedded C Programs. In: Integrated Formal Method, IFM'03. Volume 2805 of LNCS., Springer Verlag (2003) 94–113
6. Steria: Obligations de preuve: Manuel de référence. Steria - Technologies de l'information (version 3.0. Available at *http://www.atelierb.societe.com*)
7. Clearsy: B4free. Available at *http://www.b4free.com* (2004)
8. Abrial, J.R., Cansell, D.: Click'n'Prove : Interactive Proofs Within Set Theory. In et B. Wolff, D.B., ed.: 16th International Conference on Theorem Proving in Higher Order Logics - TPHOLs'2003. Volume 2758 of LNCS., Springer Verlag (2003) 1–24