Enhancing Dependability of Component-based Systems

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

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

Abstract. We present an approach for enhancing dependability of component-based software. Functionality related to security, safety and reliability is encapsulated in specific components, allowing the method to be applied to off-the-shelf components. Any set of components can be extended with dependability features by wrapping them with special components, which monitor and filter input and outputs. This approach is supported by a rigorous development methodology based on UML and the B method and is introduced on the level of software architecture.

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 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. An *adapter* is often necessary to map the provided services to the required ones.

Hence, an appropriate description of the provided and required interfaces of a software component is crucial for component-based development. In earlier papers [12, 18, 22], 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 [4], such as safety, security or reliability, can be built into 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 initial software architecture is structured in such a way that the normal behavior 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 are added, or existing components are replaced by more dependable ones, while the normal behavior remains the same. New or modified interfaces must be taken into account. In order to connect these new interfaces to the given interfaces of components, new adapters must be developed, or existing adapters must be upgraded. These adapters "shield" the components implementing the normal behavior by intercepting and possibly modifying their inputs and outputs.

In Section 2, we describe how we support component-based development using the formal specification language B. We then describe our method to integrate 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 a summary 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 the context of component-based software. We formally express provided and required interfaces using B models in order to verify their compatibility.

The Formal Method B. B is a formal software development method based on set theory, which supports an incremental development process using refinement [1]. Starting out from a textual description, a development begins with the definition of an abstract model, which can be refined step by step until an implementation is reached. The refinement of models is a key feature for incrementally developing more and more detailed models, preserving correctness in each step.

The method B has been successfully applied in the development of several complex real-life applications, such as the METEOR project [6]. 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 [7]. The B method provides structuring primitives that allow one to compose models in various ways. Large systems can be specified in a modular way and in an object-based manner [21, 20]. Proofs of invariance and refinement are part of each development. The proof obligations are generated automatically by support tools such as AtelierB [28] or B4free [13], an academic version of AtelierB. Checking proof obligations with B support tools is an efficient and practical way to detect errors introduced during development.

Specifying Component Architectures. We define component-based systems using different kinds of UML 2.0 diagrams [23]:

- Composite structure diagrams serve to express the overall architecture of the system in terms of components and their required and provided interfaces.
- Class diagrams serve to express interface data models with their different attributes and methods. An interface data model specifies the data that are passed via a given interface.

– The usage protocol of each interface can be modeled by a Protocol State Machine.
– Sequence diagrams serve to express different views of the possible interactions between different components that are connected via some interface.

Component interfaces are then specified as B models, which increases confidence in the developed systems: the correctness of the specifications, as well as correctness of the subsequent refinement process can be checked with tool support. In an integrated development process, the B models can be obtained by applying systematic derivation rules from UML to B [21, 20].

Let us give an example of a software component called TurnstileDriver, presented in Figure 1, with the B model of one of its interfaces TS_P. It represents a software driver that can lock and unlock a turnstile. This component has two interfaces, a provided one TS_P and a required one TS_R. These interfaces express that another component, connected to the TurnstileDriver, can call the lock() and unlock() methods of TS_P, but the TurnstileDriver can reciprocally call a pushed() method from the connected component when the turnstile is pushed.

```
MODEL
    TS_P
SEES
    Type
VARIABLES
    turnstile
INVARIANT
    turnstile ∈ LOCKED
INITIALISATION
    turnstile := locked
OPERATIONS
    unlock =
    PRE turnstile = locked
    THEN turnstile := unlocked
    END;
    lock =
    PRE turnstile = unlocked
    THEN turnstile := locked
    END
END
```

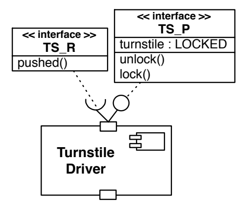

Fig. 1. TurnstileDriver

Proving Interoperability of Component Interfaces. In component-based architectures, 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 software architecture and try to show that the interfaces are compatible. Using the method B, we prove that the provided interface is a correct 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 [12].

Often, to construct a working component architecture, adapters have to be defined, connecting the required interfaces to the provided ones. An adapter is a piece of glue code that realizes the required interface using the provided interface. On the signature level, it expresses the mapping between required and provided variables. In [22], we have studied an adapter specification and its verification by giving a B refinement of the adaptation that refines the B model of the required interface and includes the provided (previously incompatible) interface.

3 Integrating Dependability Features into a Composite System

We now describe our method to integrate dependability features into a software-based system, whose software part makes use of component technology. Dependability is the ability to deliver services that can justifiably be trusted. In particular, we consider dependability properties concerning security, safety, and reliability. The latter two are relevant mainly for embedded systems, where some part of the physical world has to be controlled, whereas security is an issue also in pure data-processing systems. The basic idea of our method is to leave the normal behavior of the system untouched, and enhance dependability by

– adding dedicated components needed for realizing dependability features, or replacing used components by more dependable ones;
– constructing and/or upgrading software adapter components that connect the new "dependability components" with the existing (and unchanged) components.

In the following, we first describe the situations where our method can profitably be applied. Then, we describe how different kinds of dependability properties can be added.

3.1 Application Scenario

Our method is intended to support the following scenario. We start out with a component-based system that implements a given normal behavior, for example, controlling access of persons to a building. In the software architecture of the system, one or more components (called *application* components) can be identified that implement the normal behavior. This functionality is clearly separated from the functionality of the other components.

The components should be robust to changes: their normal behavior is to be left unchanged. Their connections to other components in terms of provided and required interfaces are not evolved. This means that enhancing dependability amounts to providing *additional* behavior that has to be executed in case of hazardous conditions, hardware or software failures, or security attacks. The system behavior in the normal case, however, remains the same. Instead of changing the components, we evolve the adapters independently of the components by providing additional functionality and dependability features.

3.2 General Procedure

Adding dependability features to a given system means to adapt the system to new requirements. The new dependability requirements may override existing requirements. For example, a functional requirement for an access control system may be that exactly the persons are admitted to a building who are authorized to be in the building. To find out if a person has permission, a database is queried. A new security requirement might state that if the database is corrupted, nobody is admitted any more, even if they are authorized according to the database. It is realized by updating existing adapters or developing new adapters. The adapters shield the application components by intercepting and possibly modifying their inputs and outputs. In general, we proceed as follows:

1. Express the new dependability requirements.
2. Express how the new requirements are related to the old ones and among each other.
3. For each dependability requirement, state what components are needed for ensuring it. Inspect the given system architecture and decide what new components are necessary, and what components must be replaced or updated.
4. Update the existing adapters and implement new dependability adapters that connect the existing components to the other components.
5. If several dependability adapters are added, it may be suitable to add one or more components that handle the new dependability-relevant events.

We use the B method for specifying component interfaces and implementing adapters. First, we can ensure that the components can indeed be plugged together as intended (see Section 2). Second, the adapter and application specifications expressed in B can be refined until code is reached.

In the following sections, we describe how to add security, safety and reliability features. We do not invent any new mechanisms but show how standard solutions for the given dependability requirements can be added to a component-based system in an incremental way.

3.3 Adding Security Features

Security is mostly concerned with confidentiality, integrity, and availability. More concrete security features concern for example authenticity and non-repudiation. In our method, availability is considered in the context of reliability, see Section 3.5.

When adding security features to a component-based system, the corresponding adapters will often implement the secure (i.e., confidential and integrity-preserving) transmission of data. Typical tasks that have to be performed include:

- checking message authentication codes to ensure integrity
- encrypt or decrypt data to ensure confidentiality
- check credentials to ensure authenticity

Existing security components may be used to realize the required security functionality.

3.4 Adding Safety Features

Safety requirements concern the reaction to hazardous situations in the environment of the system (for example, fire in a building). In these cases, the system must be put into a safe state. The safety adapters must be connected to new external components that make it possible to detect a hazardous situation. Furthermore, they must implement a transition to a safe state, because this cannot be done by the application components. What can be considered to be a safe state cannot be stated in general but depends on the specificities of the given system.

3.5 Adding Reliability Features

A standard technique to achieve reliability is to use fault-tolerance mechanisms, e.g. introducing redundant components, which can be active or passive. An active component can inform its environment when a failure occurs. In contrast, passive components just fail without informing the environment.

To realize fault tolerance with respect to failures of active components, the adapter must be able to shut down the failed component when it is informed of the failure and switch to a redundant one. In case of passive components, the adapter must check if the component works correctly, or if a failure has occurred. It must take the faulty component out of service and handle the fault, e.g. by switching to a redundant component.

4 Case Study

We illustrate our method with the case study of a simple access control system, which controls the access to a building [2]. Persons who are authorized to enter the building are equipped with a smartcard on which a user identification is stored. The access control system queries a database to obtain the information if the person is permitted to enter the building. If access is granted, a turnstile located at the entrance is unblocked, so that the person can enter the building. At the exit of the building, another turnstile is installed. It is always unblocked and only serves to count the number of persons who have left the building.

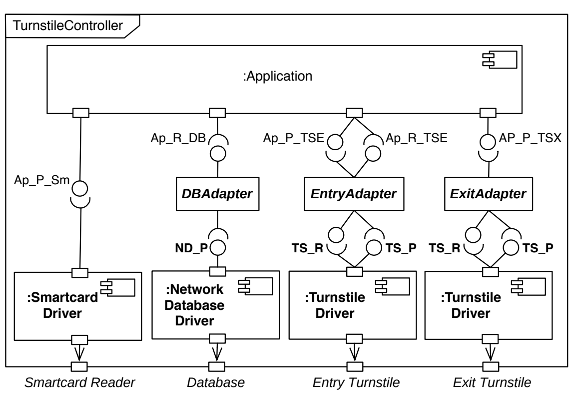

Fig. 2. Software architecture of TurnstileController

In a first version, the access control system contains no dependability features. Using our method described in Section 3, we will add two dependability features to the system by integrating appropriate new components, however leaving the basic functionality untouched. The first dependability feature concerns security. Using message authentication codes, it is checked if unauthorized modifications of the database content have occurred. In this case, the person who wants enter to the building is not admitted, and a facility service is notified. The second dependability feature concerns safety. A fire detector is added to the system. In case of fire, nobody is allowed to enter the building until the fire is dealt with, and the facility service is notified.

4.1 Architecture of the System without Dependability Features

The access control system communicates with hardware components (a smartcard reader and the turnstiles), as well as software components (the database). The controller software architecture of the access control system is shown in Figure 2, using the syntax of UML composite structure diagrams. Software components are represented as named boxes, and the interfaces between them are represented by “sockets” (required interfaces) and “lollipops” (provided interfaces).

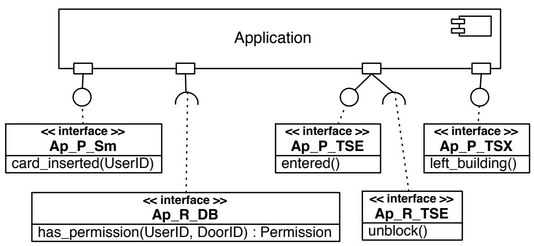

Fig. 3. The different interfaces of the Application

The software architecture of the TurnstileController is a layered one. The highest layer, i.e., the Application component, implements its normal behavior. The lowest layer consists of the software drivers that connect the software to the hardware components. A driver comes with the hardware components and should not be modified. Hence, adapters may be necessary to connect the application component to the software drivers, making up the middle layer of the architecture.

Figure 3 shows the interfaces of the Application component in more detail. For each required and each provided interface, an interface class is specified in UML notation. The interface class shows the operations belonging to the interface, together with their parameters. For example, the interface class Ap_P_Sm describes a provided interface of Application: it expresses that Application implements one method, namely card_inserted(uid), which has a user identifier uid as its parameter. This method may be called by another component connected to the interface Ap_P_Sm.

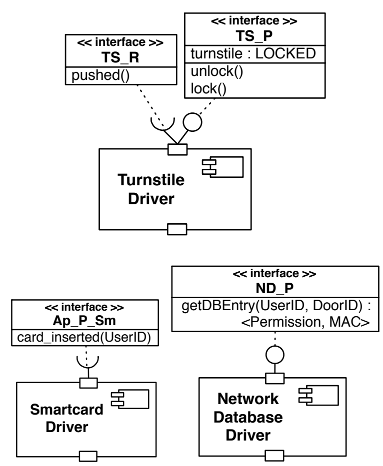

Fig. 4. Components used by the TurnstileController

The access control system uses three kinds of external components, namely a smartcard reader, a network database, and two copies of a turnstile. The corresponding drivers that control these components are named SmartcardDriver, NetworkDatabaseDriver and TurnstileDriver, respectively. Their interfaces are shown in Figure 4.

The DBAdapter As an example of an adapter, we consider the DBAdapter. Figure 5 gives a scenario of its behavior. The Application calls one of its required methods, has_permission(uid,did), which must be implemented by the DBAdapter. Parameters of the method are a user identification uid and a door identification did. As is shown in Figure 4, the database driver offers an operation getBDEntry(uid,did), which yields a permission and a message authentication code as its result. To implement has_permission(uid,did), the DBAdapter just calls the method getBDEntry(uid,did) and returns only the permission to the application component.

As Figure 3 shows, the required interface Ap_R_DB of the Application has to be implemented, i.e., an implementation of the operation has_permission(uid,did) has to be provided. This is achieved by the DBAdapter, which uses the provided interface ND_P. In Figure 6, we show how the corresponding B models are organized. To verify the correctness of the assembly, we specify a B model of the DBAdapter, which includes the B model of ND_P and refines the B model of Ap_R_DB.

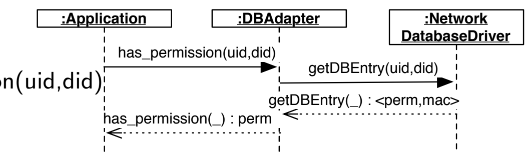

Fig. 5. Sequence diagram for the DBAdapter

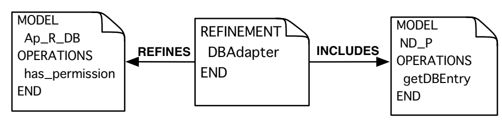

Fig. 6. B architecture for DBAdapter

4.2 Adding Dependability Features to the Access Control System

We now add two dependability features to the system, one for security and one for safety. We introduce a new component called Safety / Security / Service Application that handles security- and safety-related events by notifying the facility service. To realize the dependability features, we must introduce three new components: Secret, FacilityServiceDriver and FireDetectorDriver. Descriptions of their interfaces are given in Figure 7, and the resulting software architecture in Figure 8.

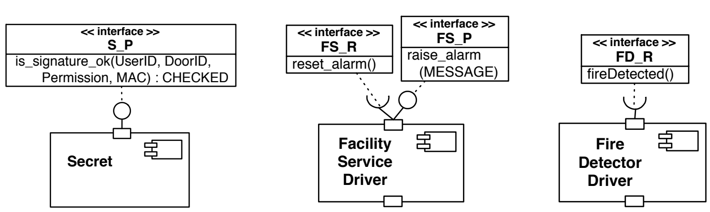

Fig. 7. New components used by the TurnstileController

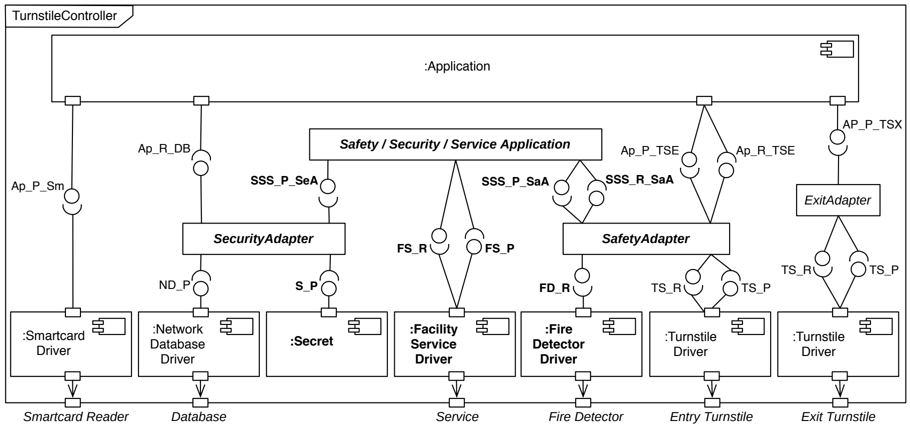

Fig. 8. Software architecture for the dependable TurnstileController

The SecurityAdapter The security feature concerns the integrity of the database. Its content is now checked using a message authentication code (MAC). A new component Secret is introduced for storing secrets that are needed to check the MAC.

The DBAdapter that connects the Application to the database is changed to use the component Secret. It is renamed to SecurityAdapter. A behavioral scenario is presented in Figure 9: the SecurityAdapter still receives a call of the method has_permission(uid, did) from the Application. It still queries the database. But now, the SecurityAdapter checks the message authentication code for each database return. In case of a violation (checked $\neq$ok), it notifies the Safety / Security / Service Application before it denies access. The Application component remains unchanged.

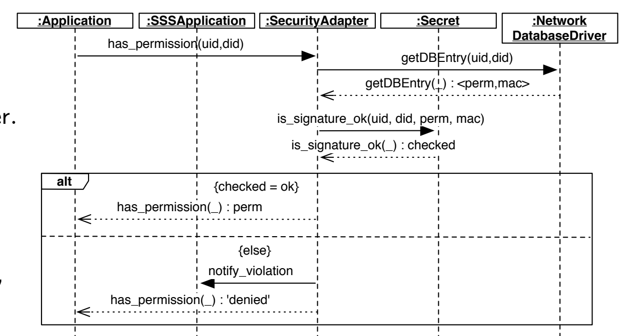

Fig. 9. Sequence diagram for the SecurityAdapter

Here, we see that the new security requirement has a higher priority than the initial functional requirement: if a manipulation of the database is detected, then the access is denied even to persons that normally have permission to enter the building.

```
REFINEMENT
    SecurityAdapter
REFINES
    Ap_R_DB
SEES
    Type
INCLUDES
    ND_P, S_P, SSS_P_SeA
OPERATIONS
    permi ←— has_permission(uid, did) =
    VAR mac, checked IN
        permi, mac ←— getDBEntry(uid, did) ;
        checked ←— is_signature_ok(uid, did, permi, mac) ;
        IF ¬(checked = ok)
        THEN
         ¬ ify_violation  ||
            permi := denied
        END
    END
END
```

Fig. 10. SecurityAdapter

The B specification of the SecurityAdapter is given in Figure 10. Again, the required interface Ap_R_DB has to be implemented, however this time not only using the provided interface ND_P, but also the provided interfaces S_P of the Secret, and SSS_P_SeA of the Safety / Security / Service Application. The OPERATIONS section contains the operation has_permission to be implemented, which is defined in terms of the operations provided by the included interfaces. Using the B models, we formally prove that the assembly correctly implements the requirements.

The SafetyAdapter The safety feature we add to the system concerns the reaction to fire. If a fire occurs, the entry turnstile must remain blocked: nobody is allowed to enter the building until the fire is extinguished (we assume the fire brigade uses another entry). Here, the EntryAdapter has to be changed to receive messages from the fire detector. It is renamed to SafetyAdapter. The SafetyAdapter blocks the entry turnstile in case of a fire and informs the Safety / Security / Service Application.

Figure 11 shows one sequence diagram concerning the SafetyAdapter, explaining the safety reaction of the adapter when it receives a fire_detected call: the turnstile will be blocked until the fire alert is canceled. Here, we see an example of how signals from the application component are intercepted: the unblock signals of the Application are not passed on to the entry turnstile; hence, it remains blocked.

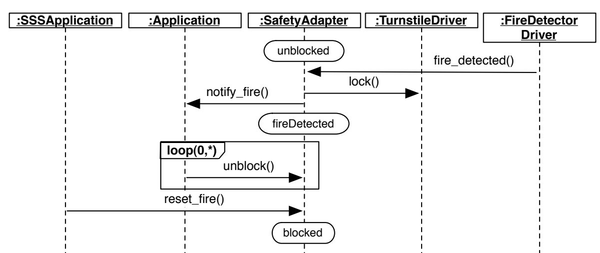

Fig. 11. A sequence diagram for the SafetyAdapter

Figure 12 shows how the SafetyAdapter is specified in B. It implements the interfaces SSS_R_SaA, Ap_R_TSE, FD_R, and TS_R, using the interfaces SSA_P_SaA, Ap_P_TSE, and TS_P. The B model called SafetyAdapter_abs is needed for technical reasons: a B model can only refine a single B model and not several ones.

We do not describe the new application component Safety / Security / Service Application in detail. It serves to pass on a security or safety alarm to the facility service, and it receives a message from the facility service when the alarm is canceled.

```
REFINEMENT
    SafetyAdapter
REFINES
    SafetyAdapter_abs
INCLUDES
    TS_P_U, Ap_P_TSE, SSS_P_SaA
SEES
    Type
VARIABLES
    entry
INVARIANT
    entry ∈ ENTRY_STATES ∧
    ( turnstile = locked ⇒
        entry ∈ {blocked, fireDetected}) ∧
    ( turnstile = unlocked ⇒
        entry = unblocked)
INITIALISATION
    entry := blocked
OPERATIONS
    unblock =
    IF
        entry = blocked
    THEN
        unlock ;
        entry := unblocked
    END ;
    pushed =
    IF
        entry = unblocked
    THEN
        entered ;
        lock ;
        entry := blocked
    END ;
    fire_detected =
    IF
        entry ≠ fireDetected
    THEN
        IF ( turnstile = unlocked) THEN lock END ;
        ¬ ify_fire ;
        entry := fireDetected
    END ;
    reset_fire =
    IF
        entry = fireDetected
    THEN
        entry := blocked
    END
END
```

Fig. 12. SafetyAdapter

With this case study, we have shown how dependability features can be added to a component-based system in a modular manner. Other dependability features could be added to the access control system in the same way. Examples are an authentication mechanism for the smartcard interface, a redundant arrangement of fire detectors, or checking for memory errors.

5 Related Work

A lot of studies have already been done on component-based approaches. Beugnard et al. [8] propose to define contracts for components, distinguishing four levels of contracts: syntactic, behavioral, synchronization, and quality of service. They do not introduce data models for interfaces, and it cannot easily be checked if two components can be combined. Roshandel and Medvidociv [26] propose to specify four different views of software components, namely the interface, static behavior, dynamic behavior, and interaction protocol views. To ensure dependability, the consistency of the different views is checked. Cheesman and Daniels [11] propose a process to specify component-based software, which starts with an informal requirements description and produces an architecture showing the components to be developed or reused, their interfaces and their dependencies.

Canal et al. [10] use a subset of the polyadic π-calculus to deal with component interoperability at the protocol level. The limitation of this approach is the low-level description of the used language and its minimalistic semantics. Bastide et al. [5] use Petri nets to specify the behavior of CORBA objects, including operation semantics and protocols. The difference to our approach is that we take into account the invariants of the interface specifications. Zaremski and Wing [31] 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 [17, 29] have also proposed to enrich component interface specifications by providing information at signature, semantic and protocol levels. Henzinger and Alfaro [3] propose an approach allowing the verification of interfaces interoperability based on automata and game theories, which is well suited for checking the interface compatibility at the protocol level.

Concerning component adaptation, several proposals have already been made. Some practice-oriented studies analyze different issues when adapting of third-party components [16]. A formal foundation of the notions of interoperability and component adaptation is set up in [30]. Component behavior specifications are given by finite state machines.

Braccalia et al. [9] specify an adapter as a set of correspondences between methods and parameters of the required and provided components. The 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 Schmidt present adapters in the context of concurrent systems. They consider a certain class of protocol interoperability problems and generate adapters for bridging component protocol incompatibilities, using interface described by finite parameterized state machines [27].

In contrast to the above approaches, we prefer to use the B method, because it allows us to not only consider component compatibility at the protocol level, but also at the signature and semantic levels, and because of its tool support.

A general approach to wrappers for common security concerns is described in [15]. Popov et al. [24] show that wrappers are components that monitor and ensure the non-functional properties at interfaces between components. They improve dependability by adding fault tolerance. In [14], the authors propose to structure fault-tolerant component-based systems that use off-the-shelf components, at the architectural level, using constructs similar to the multi-versioning connector [25].

In contrast to the above approaches, our method stresses the methodological aspects of evolving a given component-based system to make it more dependable. In an earlier paper [19], we have addressed to problem of adding features to component-based systems. But there, we did not use the B method, and the newly integrated features did not concern dependability, but the addition of new functionality.

6 Conclusion

The success of the component-construction paradigm in mechanical and electrical engineering has led to calls for its adoption in software development. We have described a method to integrate dependability features into a composite system. We start from an initial software architecture describing the the normal behavior of the system for. Dependability is then enhanced in an incremental way, by modifying adapter components and possibly adding new adapter or new application components.

Using the formal method B and its refinement and assembling mechanisms to model the component interfaces and the adapters, we pay special attention to the question of guaranteeing the interoperability between the different components. The B prover guarantees that the adapter is a correct implementation of the required functionalities in terms of the existing components. With this approach, the verification of the interoperability between the connected components is achieved at the signature, the semantic and the protocol levels. In summary, the advantages of our approach are the following:

• Dependability features can be integrated one by one, as needed.

• The necessary changes to the software architecture are local; the functionality for the normal case is not changed.

• Components and dependability features can be further evolved independently of each other.

• Our method gives guidance on how the addition of dependability features can be performed in a systematic way.

• Using B, it can be checked that the components of the evolved software architecture indeed interoperate as intended.

• The B specifications of the new or evolved software components can be used as the starting point of an implementation. For this purpose, the B refinement mechanism can be used.

In this way, we have proposed a "dependable" process for making component-based systems more dependable.

References

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

[2] 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.

[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. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. on Dependable and Secure Computing*, 1(1):11–33, Jan. 2004.

[5] 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.

[6] 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.

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

[8] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *IEEE Computer*, pages 38–45, July 1999.

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

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

[11] J. Cheesman and J. Daniels. *UML Components – A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2001.

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

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

[14] P. A. de Guerra, C Mary, F. Rubira, A. Romanovsky, and de Lemos R. A fault-tolerant software architecture for COTS-based software systems, 2003.

[15] C. Fetzer and Z. Xiao. HEALERS: A Toolkit for Enhancing the Robutness and Security of Existing Wrappers. In *Proc. International Conference on Dependable Systems and Networks*, 2003.

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

[17] J. Han. A comprehensive interface definition framework for software components. In *The 1998 Asia Pacific software engineering conference*, pages 110–117. IEEE Computer Society, 1998.

[18] D. Hatebur, M. Heisel, and J. Souquières. A method for component-based software and system development. In *Proceedings of the 32tnd Euromicro Conference on Software Engineering And Advanced Applications*, pages 72–80. IEEE Computer Society, 2006.

[19] M. Heisel and J. Souquières. Adding features to component-based systems. In M.D. Ryan, J.-J. Ch. Meyer, and .-D. Ehrich, editors, *Objects, Agents and Features*, LNCS 2975, pages 137–153. Springer-Verlag, 2004.

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

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

[22] I. Mouakher, A. Lanoix, and J. Souquières. Component adaptation: Specification and verification. In R. Reussner W. Weck and C. Szyperski, editors, *11th International Workshop on Component Oriented Programming (WCOP'06)*, pages 23–30, 2006.

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

[24] P. Popov, L. Strigini, S. Riddle, and A. Romanovsky. Protective Wrapping of OTS components. In *4th ICSE WWorkshop on Component-Based Software Engineering: Component Certification and System Prediction*, 2001.

[25] M. Rakic and N. Medvidovic. Increasing the confidence in off-the-shelf components. In *Symposium on Software Reusability*, pages 11–18, 2001.

[26] R. Roshandel and N. Medvidovic. Multi-view software component modeling for dependability. In R. de Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems II*, LNCS 3069, pages 286–306. Springer Verlag, 2004.

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

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

[29] A. Vallacillo, J. Hernandez, and M. Troya. Object interoperability. In *Object Oriented Technology: ECOOP'99 Workshop Reader*, pages 1–21, 1999.

[30] D. D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst.*, 19(2):292–333, 1997.

[31] A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, 1997.