A Method for Component-Based Software and System Development

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

[1] Universität Duisburg-Essen, Institut für Medientechnik und Software Engineering
D-47048 Duisburg, Email: {denis.hatebur, 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 a method for component-based software and system development, where the interoperability between the different components is given special consideration. The method uses existing notations and languages with their associated tools: context diagrams for analyzing and structuring the problem, composite structure diagrams for describing the overall system in terms of components and interfaces, sequence diagrams to describe the behavior of each component, and the formal method B for specifying the interfaces of the different components and for proving their interoperability. The method proposes to integrate these different notations; at the end of the process, the interoperability is guaranted by the use of the B method with its underlying concept of refinement, and its powerful tool support, the B prover.**

Keywords: Component based approach, interoperability, formal method

I. Introduction

The idea underlying the paradigm of component orientation [13], [20] is to develop software systems not from scratch but by assembling pre-fabricated parts, as is common in other engineering disciplines. As in object orientation, components are encapsulated, and their services are accessible only 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. These considerations lead to the following requirements on how to describe components in such a way that they can be assembled into systems in a systematic way:

- The specification of a component must contain sufficient information to decide whether or not to acquire it for integration in a new 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 *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).
- The interface operations provided or required by some component usually cannot be called in an arbitrary order. Instead, certain communication protocols must be adhered to. These protocols also must be provided in a comprehensive component description.

The method for component-based software and system development we present in this paper was designed to fulfill these requirements. It is based on existing languages and notations with their associated tools to appropriately describe existing components or components to be constructed, and to check if two components can be connected via given interfaces or not. In particular, we use the following languages:

- Jackson's context diagrams [15] serve to analyze and structure the problem in terms of domains and shared phenomena (which will be organized into interfaces later).
- UML 2.0 [22] composite structure diagrams serve to express the overall architecture of the system.
- UML 2.0 sequence diagrams serve to express the visible behavior of components. They show communication sequences with other connected components.
- The formal method B [1] serves to specify interfaces. An interface specification consists of a data model and the specification of the different operations provided or required by the interface. Since the B concepts of a machine and of refinement fit well with components and their interoperability, we use a B tool for proving component interoperability. (Using for example the object constraint language OCL [24], and generating verification conditions from scratch would be much more tedious.)

In the following, we present an overview of the B method in Section II. We then describe our method in Section III and illustrate it with the case study of an access control system in Section IV. In Section V, we discuss related work. The paper finishes with some concluding remarks in Section VI.

II. The Formal Method B

The B method [1] 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 refinement of models is a key feature for incrementally developing models from textual descriptions, preserving correctness in each step. Thus, B follows the proof-based development paradigm [19]. 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. An example of a B machine is given Fig. 1. The invariant clause characterizes the sensible 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.

MACHINE
T_P_C
SETS
Turnstile_States $=$ $\{locked,\ unlocked\}$
VARIABLES
state
INVARIANT
state $\in$ *Turnstile_States*
INITIALISATION
state $:=$ *locked*
OPERATIONS
unlock_c $\hat{=}$
PRE *state* $=$ *locked*
THEN *state* $:=$ *unlocked*
END;
lock_c $\hat{=}$
PRE *state* $=$ *unlocked*
THEN *state* $:=$ *locked*
END
END

Fig. 1. An example of a B Specification

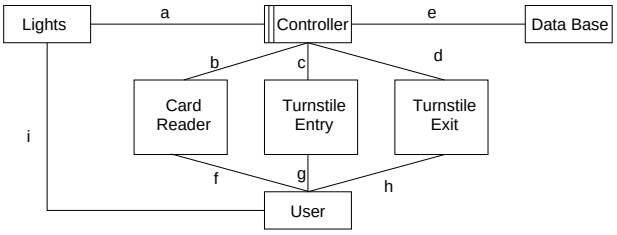
The B method provides structuring primitives that allow one to compose machines in various ways. Large systems can be specified in a modular way and in an object-based manner [18], [17]. Proofs of invariance and refinement are part of each development. The proof obligations are generated automatically by support tools such as AtelierB [19] or B4free [11], an academic version of AtelierB. Checking proof obligations with B support tools (either through automatic or interactive proofs) [2], is an efficient and practical way to detect errors introduced during development.

III. A Method for Component-Based Development

Our goal is to provide a method for component-based software and system development that pays special attention to the question 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. Our method consists of the following steps.

A. Set up a context diagram

Context diagrams as introduced by Jackson [15] serve to analyze and structure a given development problem. They consist of rectangles that are connected by lines (for an example, see Fig. 2). The rectangles denote *domains*, and a connecting line between two (or more) domains indicates that there are *shared phenomena* between the connected domains. Shared phenomena may be events, operation calls, messages, and the like. They are observable by at least two domains, but controlled by only one domain. For example, if a user pushes a turnstile, this is a phenomenon shared by the user and the turnstile, which is controlled by the user. For each connecting line, the corresponding shared phenomena are given in a context diagram.



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!{block, unblock}
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!{push} TE! {locked, unlocked}
h: U!{push}
i: L!{see_green, see_red}

Fig. 2. Context diagram for the access control system

In the original version of context diagrams, each such diagram contains exactly one *machine domain*, denoted by a rectangle with a double vertical stripe. This is the domain that has to be constructed to solve the problem. All other domains are *given domains* that already exist[1]. Their properties (called *domain knowledge*) have to be taken into account in the problem solving process.

In our method, domains correspond to components or actors, and connecting lines and their shared phenomena correspond to interfaces between components or components and actors. In contrast to Jackson, we allow several or even zero machine domains in a context diagram. If a component has to be constructed, it is expressed as a machine domain. If a

[1]There are also *designed domains*, which are data structures to be developed. We do not use designed domains in this work; hence, they will not be mentioned further.

component exists, it is expressed as a given domain. Thus, context diagrams with no machine domain express the fact that a system shall be assembled from existing components only, and that we have to check the interoperability of these components. If a context diagram contains at least one machine domain, we have more flexibility in the development process. For example, we might change the interfaces of a component to be constructed if this is necessary in order to interoperate with the given components.

Hence, the domains contained in our context diagrams may play one of the following roles:

- A machine domain corresponds to a software component to be constructed, (e.g., *Controller* in Fig. 2).
- A given domain can be
  - an existing software component to be used (e.g., *Data Base* in Fig. 2)
  - an existing hardware component to be used (e.g., *Turnstile Entry* in Fig. 2)
  - an actor communicating with the system (e.g., *User* in Fig. 2)

Note that for existing components we need not distinguish between hardware and software components, as both are described in the same way. Thus, our method not only covers component-based *software* development, but also component-based *system* development.

A second generalization of Jackson's context diagrams concerns the control of shared phenomena. In Jackson's method, this information is only added in a later step. In contrast, we add it to the context diagram. The notation $U!\{push\}$ means that the *User* component controls the shared phenomenon *push*.

Setting up a (generalized) context diagram, we structure the given development problem by identifying the different actors and the components that are available or must be constructed. Specifying the shared phenomena between the components is a preparation for the next step of the method, where all provided and required interfaces must be identified.

B. Construct the system- or software architecture

Based on the context diagram developed in the first step of the method, we now decide on the provided and required interfaces of all the identified components. For this purpose, we have to inspect the shared phenomena expressed in the context diagram, and consider the following rules:

- A shared phenomenon observable but not controlled by component *C* corresponds to an operation or signal in a *provided* interface of *C*, because *C* must in some way be notified of the occurrence of the shared phenomenon.
- Conversely, a phenomenon controlled by component *C* will often correspond to an operation or signal in a *required* interface of *C*, because controlling a phenomenon often corresponds to sending a signal or message to another component (thus invoking an operation of a provided interface of the receiver component).
  However, there are exceptions to this rule. They concern *passive* components that are mere data structures. Such components do not send messages to their environment without an external stimulus. Hence, the phenomena controlled by passive domains correspond to return values of operations of a provided interface. An example is the data base in Fig. 2. Although it controls the phenomenon *authorized*, it only has a provided interface, where the *authorized* phenomenon is the return value of a data base query issued by the controller component.

The software or system architecture is expressed as a UML 2.0 composite structure diagram. Such diagrams contain named rectangles, called *parts*. These parts are the components of the system. Parts may have *ports*, denoted by small rectangles, and ports may have interfaces associated to them. Interfaces may be required or provided. Provided interfaces are denoted using the "lollipop" notation, and required interfaces using the "socket" notation. For an example, see Fig. 3.

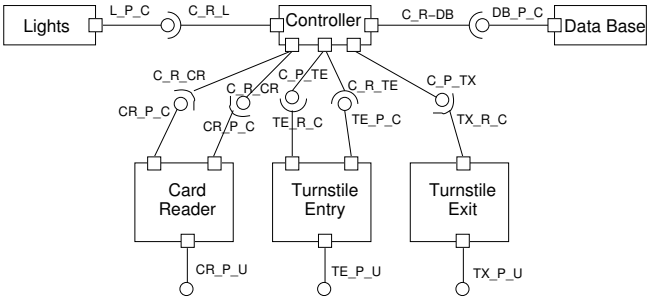


Fig. 3. Architecture of the global access control system

The procedure for setting up the architectural diagram for the system is as follows. Each connecting line between two domains of the context diagram must be transformed into one or two connections between two components. Such a connection consists of a provided interface for one of the components and a corresponding required interface of the other component. For a bi-directional communication, we need two connections (e.g., between *Turnstile Entry* and *Controller* in Fig. 3); for a one-directional communication, we only need one connection (e.g., between *Data Base* and *Controller* in Fig. 3). The decision on provided and required interfaces must be taken according to the rules given above. Finally, we delete all domains corresponding to actors from the diagram.

C. Specify components

For each component of the architecture, a specification must be set up containing:

- Sequence diagrams describing the visible behavior of the specified component; the sequence diagrams must contain all operations or signals of all interfaces and all other components the specified component is connected with. Messages received by the specified component from its environment must correspond to an operation or signal of some provided interface, while messages sent out by the specified component must correspond to an operation or signal of some required interface.
- 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 (IDM, see Section I) must be the same, and the IDM must be encoded in the B machine by 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

When a component manipulates data, it is possible to use a UML class diagram to express the interface data model for reasons of readability. This class diagram can then automatically be transformed into a B specification [18].

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. Both pre- and postcondition will refer to the interface data model.

The behavioral specification and the operation specifications must be coherent. To ensure coherence, the sequence diagrams should be annotated with states, which are also used to express the pre- and postconditions of the B operations. In this way, we make sure that the communication sequences required by the sequence diagrams do not violate any preconditions of interface operations.

Such a specification contains all the information that is needed to decide if a given component can be used in a given context or not.

D. Prove interoperability

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 the 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 shown in the architectural diagram. The process of proving interoperability between components is described in [9].

If we cannot demonstrate the interoperability of an intended connection of the architectural diagram, we either can try to change the specification of a component to be developed (corresponding to a machine domain in the context diagram), or we can try to develop an adapter. In both cases, the architectural diagram and the component specifications will have to be adjusted accordingly.

If the latter approach is not possible either, we have to conclude that the components cannot be connected as intended.

In the case where all components are given, the result of our method is a proof that the system can be assembled as specified in the architectural diagram. In the case where some components do not yet exist but must be constructed, our method yields detailed specifications of all the components to be developed.

IV. Case study: Access Control System

We illustrate our method with the case study of a simple access control system. The access to a building is to be controlled. Persons (called users) who are authorized to enter the building have access cards with some identification 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.

A user who wants to enter the building inserts his or her card into the card reader. The information on the card is read, and a data base is queried to decide if the user is granted access or not. If access is granted, the green light is turned on for some time, the card is ejected, and the entry turnstile – which is normally blocked – is unblocked. The entry turnstile is re-blocked either after the user has entered or after some timeout. If the user does not take the card within 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 card is ejected. Again, if the user does not take the card within some time limit, the card is retracted and kept.

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.

A. Context diagram

A context diagram for this access control system is shown in Fig. 2. We have one component to be developed, namely *Controller*, and one domain corresponding to an actor, namely *User*. The other domains correspond to existing components.

As an example, we consider the interfaces *c* and *g*: the controller can give commands to the entry turnstile to block or unblock it. Conversely, the entry turnstile sends a signal *entered* to the controller. This signal is sent as a reaction of a *push* event caused by the user (interface *g*). Since the user can observe if the turnstile is locked or not, the interface contains two phenomena *locked* and *unlocked* controlled by the entry turnstile.

B. Architecture of the system

The architecture of the system is given in Fig. 3. We have used the following naming conventions for interfaces. Each interface name has the form X_Y_Z, where

- X is an abbreviation of the name of the component the interface belongs to, e.g., its first letter;
- Y is either “P” for provided or “R” for required;
- Z is the abbreviation of the name of the component the given component is connected to.

As an example, *C_P_TE* is the provided interface of the *Controller* component which is connected to the *Turnstile Entry* component.

Note that the *User* is no longer contained in the diagram, and that the data base only has a provided interface, because it is a passive component (even though it controls the phenomenon *authorized*). In all other interfaces, controlled phenomena correspond to operations in required interfaces, and observed phenomena correspond to operations in provided interfaces.

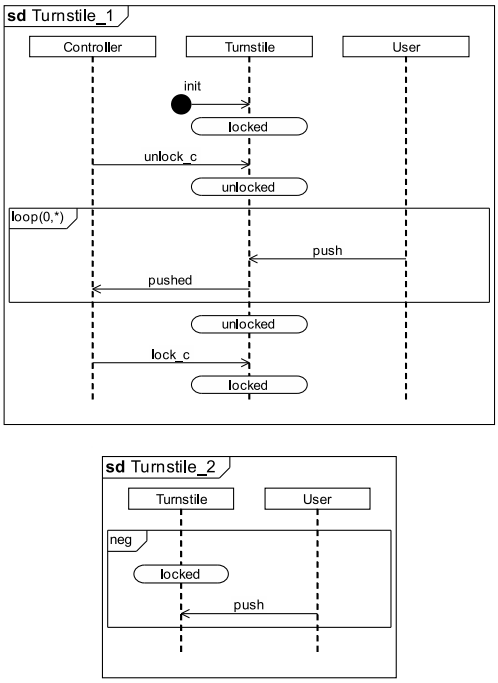


Fig. 4. Behavioral specification of the turnstile component

C. Component Specifications

We want to use two copies of the same turnstile component available on the market to be installed as an entry and an exit turnstile, respectively. For reasons of space, we only present the specification of that turnstile component, and parts of the controller specification.

1) Specification of the turnstile component: The available turnstile component has the following interfaces:

- *T_P_C*, its provided interface to a controller component, with two operations *lock_c* and *unlock_c*;
- *T_R_C*, its requested interface to the controller component, with an operation *pushed*;
- *T_P_U*, its provided interface to an external user, with an operation *push*.

There is no required interface to the user because the user finds out if the turnstile is locked by trying to push it and not be being sent a message from the turnstile.

MACHINE
T_R_C
SETS
Turnstile_States $= \{locked, unlocked\}$
VARIABLES
state
INVARIANT
state $\in$ *Turnstile_States*
INITIALISATION
state $:=$ *locked*
OPERATIONS
pushed $\widehat{=}$
PRE *state* $=$ *unlocked*
THEN SKIP **END**
END

Fig. 5. Specification of *T_R_C*

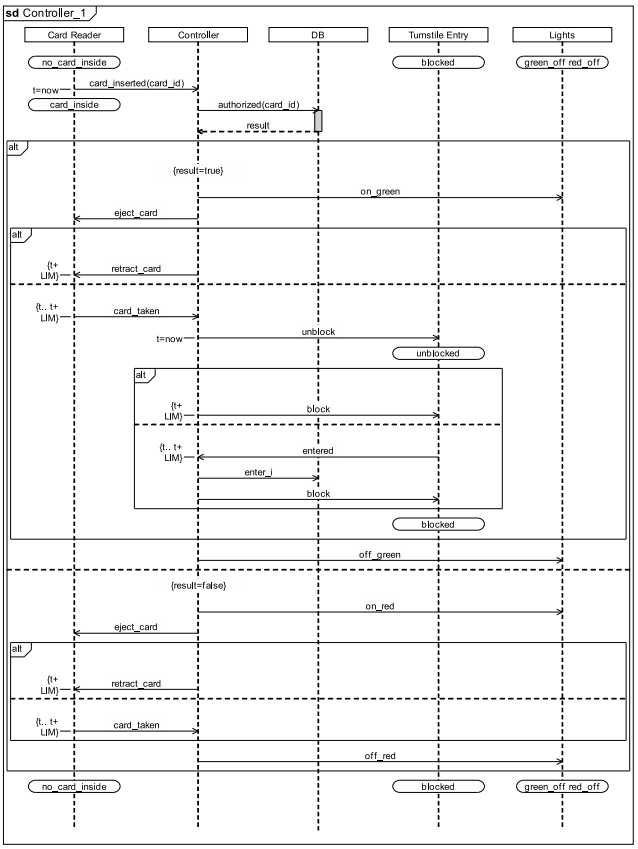


Fig. 6. Sequence diagram for the controller component (entry)

The turnstile component has two states, namely *locked* and *unlocked*, and its initial state is *locked*. It has no invariant property. The externally visible behavior, i.e., its usage protocol, is given in Fig. 4. The first sequence diagram expresses that initially the turnstile is locked. After it receives an *unlock_c* command from the controller, it changes its state to *unlocked*. In that state, an arbitrary number of *push* events can be received from the user. When the turnstile component receives a *lock_c* command from the controller, it re-enters the state *locked*. The second sequence diagram expresses the fact that in state *locked*, no *push* events can be received from the user. The *neg* construct has been used to express this forbidden scenario. Note that the given turnstile component is specified using different names for the states and the operations than the ones used in Fig. 2, because it cannot be assumed that component consumers and component producers choose the same names independently of each other.

We now give B specifications for two of the tree interfaces. The interface data models of interfaces that connect the same two components must be the same. Hence, *T_P_C* and *T_R_C* have the same IDM, including the state of the turnstile with the two possible values, *locked* and *unlocked*, and the initial state *locked*. The interface specifications are shown in Figs. 1 and 5. The operations *lock_c* and *unlock_c* change the state of the turnstile, whereas the operation *pushed* corresponds to sending a signal to the controller and does not change the state of the turnstile. As specified in the sequence diagram,

it can only be invoked if the turnstile is in state *unlocked*.

2) Specification of the controller component: In the same way, we have to specify the controller component. Since the controller is connected with five other components, its behavioral specification is much more complex than the behavioral specification of the turnstile. Fig. 6 shows the behavior of the controller when a user enters the building. It captures exactly the behavior sketched in the informal description of the access control system. Figure 7 shows the behavior of the controller component when a user leaves the building.

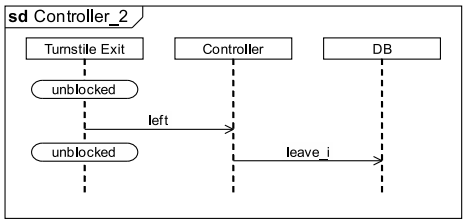


Fig. 7. Sequence diagram for the controller component (exit)

As an example of an interface specification, we give the B specification of the required interface of the controller with the entry turnstile, *C_R_TE* in Fig. 8. It looks quite similar to the specification of the interface *T_P_C*, and in fact, we will show that the two interfaces can be connected. The specification of interface of the controller with the exit turnstile is given in Fig. 9. Note that here we require that the turnstile is always unblocked, as specified in the behavioral specification, see Fig. 7.

MACHINE
C_R_TE
SETS
Entry_States_Turnstile $= \{blocked, unblocked\}$
VARIABLES
e_state
INVARIANT
e_state $\in$ *Entry_States_Turnstile*
INITIALISATION
e_state $:=$ *blocked*
OPERATIONS
unlock $\hat{=}$
PRE *e_state* $=$ *blocked*
THEN *e_state* $:=$ *unblocked* **END;**
lock $\hat{=}$
PRE *e_state* $=$ *unblocked*
THEN *e_state* $:=$ *blocked* **END**
END

Fig. 8. Specification of *C_R_TE*

D. Proving Interoperability

We now must prove that the controller can be connected with two of the given turnstile components as specified in Fig. 3. For this purpose, we must carry out three refinement proofs. As an example, we show that a transformed version of the interface *T_P_C* is a refinement of the interface *C_R_TE*, see Fig. 10. The refinement machine *NewT_P_C* is a transformation of the machine *T_P_C* (Fig. 1), where *unlock_c* is renamed to *unlock* and *lock_c* is renamed to

MACHINE
C_P_TX
SETS
Exit_States_Turnstile $= \{blocked, unblocked\}$
VARIABLES
x_state
INVARIANT
x_state $\in$ *Exit_States_Turnstile*
INITIALISATION
x_state $:=$ *unblocked*
OPERATIONS
left $\hat{=}$
PRE *x_state* $=$ *unblocked*
THEN SKIP **END**
END

Fig. 9. Specification of *C_P_TX*

lock. This renaming is necessary because the definition of refinement in B requires that the refining machine defines operations with the same names as the refined machine. The *linking invariant*, *state* $=$ *locked* $\Leftrightarrow$ *e_state* $=$ *blocked*, relates the states of the different machines.

REFINEMENT
New_T_P_C
REFINES
C_R_TE
SETS
Turnstile_States $= \{locked, unlocked\}$
VARIABLES
state
INVARIANT
state $\in$ *Turnstile_States* $\wedge$
(*state* $=$ *locked* $\Leftrightarrow$ *e_state* $=$ *blocked*)
INITIALISATION
state $:=$ *locked*
OPERATIONS
unlock $\hat{=}$/* *unlock_c* */
PRE *state* $=$ *locked*
THEN *state* $:=$ *unlocked* **END;**
lock $\hat{=}$ /* *lock_c* */
PRE *state* $=$ *unlocked*
THEN *state* $:=$ *locked* **END**
END

Fig. 10. Specification of $New_T\_P\_C$

We might now try to perform a similar refinement proof for the interfaces *T_R_C* and *C_P_TX*, i.e., to show that the controller and the exit turnstile can be connected as intended, see Fig. 11. However, this proof fails, because the initializations of the two machines are incompatible.

The problem is that the given turnstile component comes in state *locked*, but to be usable as an exit turnstile in the access control system, it must be in state *unlocked*. Hence, we must change the specification of the controller. The controller must get a new interface *C_R_TX* which it can used to initialize the turnstile component. As an initialization operation, the operation *unlock_c* provided by the turnstile component can be used. Figures 12 and 13 show the new interface of the controller and its interoperability with the interface *T_P_C* of the turnstile component.

REFINEMENT
New_C_P_TX
REFINES
T_R_C
SETS
$Exit\_States\_Turnstile = \{blocked, unblocked\}$
VARIABLES
x_state
INVARIANT
$x\_state \in Exit\_States\_Turnstile \wedge$
$(x\_state = blocked \Leftrightarrow state = locked)$
INITIALISATION
$x\_state := unblocked$
OPERATIONS
pushed $\hat{=}$ /* *left* */
PRE $x\_state = unblocked$
THEN SKIP **END**
END

Fig. 11. Specification of $New_C\_P\_TX$

MACHINE
C_R_TX
SETS
$Exit\_States\_Turnstile = \{not\_init, init\}$
VARIABLES
x_state
INVARIANT
$x\_state \in Exit\_States\_Turnstile$
INITIALISATION
$x\_state := not\_init$
OPERATIONS
initialize $\hat{=}$
PRE $x\_state = not\_init$
THEN $x\_state := init$ **END**
END

Fig. 12. Specification of *C_R_TX*

These changes have to be propagated to the other specifications as follows:

- The initialization of the machine *C_P_TX* must be changed, so that it also allows an initial state *locked*.
- A connection between the controller and the exit turnstile must be added to the architectural diagram. It connects the interfaces *C_R_TX* and *TPC*.
- The initialization of the exit turnstile must be achieved by the controller. The corresponding sequence diagram is shown in Fig. 14.

Thus, we have shown how our method supports developers in assembling systems from components, always guaranteeing that the different components are able to interoperate in the intended way.

V. 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. 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.

REFINEMENT
Init_T_P_C
REFINES
C_R_TX
SETS
$Turnstile\_States = \{locked, unlocked\}$
VARIABLES
state
INVARIANT
$state \in Turnstile\_States \wedge$
$(state = locked \Leftrightarrow x\_state = not\_init)$
INITIALISATION
$state := locked$
OPERATIONS
initialize $\hat{=}$/* *unlock_c* */
PRE $state = locked$
THEN $state := unlocked$ **END**
END

Fig. 13. Refinement $InitT\_P\_C$ of *C_R_TX*

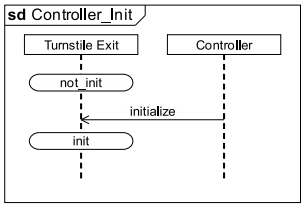


Fig. 14. Sequence diagram for initialization of the exit turnstile

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 [21]. 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. However, they do not introduce data models for their interfaces. Hence, 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 in-

terfaces and their dependencies. For each interface operation, a specification is developed, consisting of a precondition, a postcondition and possibly an invariant.

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 [25] 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], [23] 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].

VI. Conclusion

We have presented a method for component-based software and system development. In this method, components are considered as black boxes. They are only described by their visible behavior and by their interfaces. This approach makes it possible to describe hardware and software components as well as existing components and components to be constructed in the same way. Interoperability between component is defined rigorously and can be checked with tool support. Furthermore, methodological guidance is given to developers, as our method consists of four well-defined steps. Thus, the ideas underlying the concept of component-based development can be fully exploited, and the way of constructing systems becomes more similar to other engineering disciplines.

To construct a working system out of components, adapters have to be defined that implement the transformation of required interface data into provided interface data and vice versa. In some cases, we can relax compatibility conditions and use more liberal versions of specification matching, e.g. plug-in-post matching [25]. In this case, an adapter must check if the precondition of the provided operation holds. If not, it has to take appropriate actions other than calling the provided operation. Thus, the construction of adapters becomes a program synthesis problem. This problem becomes more complex for weaker versions of specification matching. We are currently working on alternative versions of compatibility and their mappings to refinement in B, and to give patterns for the corresponding adapters.

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 H. R. Arabnia and H. Reza, editors, *International Worshop on Formal Aspects on Component Software*. CSREA Press, 2005. To appear in ENCTS 2006.

[10] 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.

[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] 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.

[19] Steria. *Obligations de preuve: Manuel de référence*. Steria - Technologies de l'information, version 3.0. Available at *http://www.atelierb.societe.com*.

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

[21] K. Turowski, editor. *Standardized Specification of Business Components*. Gesellschaft für Informatik, 2002.

[22] UML Revision Task Force. *OMG Unified Modeling Language: Superstructure*, August 2005. `http://www.uml.org`.

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

[24] J. Warmer and An. G. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.

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