Jörg Ackermann

Using a Specification Data Model for Specification of Black-Box Software Components

Compositional plug-and-play-like reuse of black-box components requires sophisticated techniques to specify components. In the past little attention has been paid to the use of a conceptual data model to guide behavioral specification of components. In this paper we show how specifications are limited if no such data model is used and overcome these limitations by introducing the concept of a component specification data model. Additionally we propose a solution on how to obtain interface specification data models that are integrated and consistent with the component specification data model.

1 Motivation

Combining off-the-shelf software components offered by different vendors to customer-individual business application systems is a goal that is followed-up for a long time [McIl68]. Applying this approach promises (amongst others) a shorter time to market, increased adaptability and reduced development costs ([Brow00], [SzGM02]). Composing software components developed independently from each other to business application systems requires that the components are compatible or that compatibility can be reached e.g. by adaptation. (If compatibility is not attainable then different components must be selected.) This requires the compatibility of the core data models of all employed components. Compatibility would be increased and necessary adaptation reduced if the components were based on a common data or object model.

To be reused successfully a component must provide all required information in form of a component specification. A precise and reliable specification of software components supports sound selection and trust in its correct functioning [GeGh06]. Moreover, component specifications are a prerequisite for the success of component markets [HaTu02] as well as for a composition methodology and tool support [Over04]. For these reasons the specification of software components is a critical success factor for component-based development of information systems.

Many existing specification approaches use pre- and postconditions to specify behavioral aspects of black-box components (see Sect. 2). Here the question arises if the component specification should be equipped with some kind of specification data model or if the behavioral specification should be based alone on the interface specification. So far the use of such a model is not wide-spread: Most publications about black-box component specification do not discuss this issue and implicitly decide not to use a model – cf. e.g. [BJPW99], [Turo02] and [Over04]. Only some of our own publications ([AcTu03], [Acke05], [AcTu06]) employ such a model but do not sufficiently explain and justify its application. Moreover, the justification of the approach was in the past frequently questioned – the claim being that such a model does not adhere to black-box principles. We do not agree with this argument and claim instead that specification data models promote a simpler and more expressive behavioral specification of components.

The contribution of this paper is the introduction of specification data models in black-box component specification: After a discussion on the relevance of contracts for black-box components (Sect. 3) we show how specifications are limited if no data model is used, develop the concept of component specification data models and discuss the advantages of its application (Sect. 4). Additionally we propose a solution how to obtain interface specification data models that are integrated and consistent with the

component specification data model (Sect. 5). In the related work section we discuss in detail how other specification approaches deal with the question of data models and what limitations they have(Sect.6). The paper concludes with a summary (Sect. 7).

2 Specification of Black-Box Software Components

The comprehensive and standardized specification of black-box software components is a critical success factor for building component-based information systems. With *specification* of a component we denote the complete, unequivocal and precise description of its external view – that is which services the component provides under which conditions [Turo02]. Currently there exists no generally accepted and supported specification standard covering all aspects relevant to component-based software engineering (CBSE). Various authors addressed specifications for specific tasks of the development process as e.g. design and implementation ([DSWi98], [ChDa01]), component adaptation [YeSt97] or component selection [HeLi01]. Approaches towards a comprehensive specification of black-box components are few and include the four-layer model of [BJPW99], the memorandum for standardized specification of business components [Turo02] and the standardized framework for specification of software components (UnSCom) [Over04].

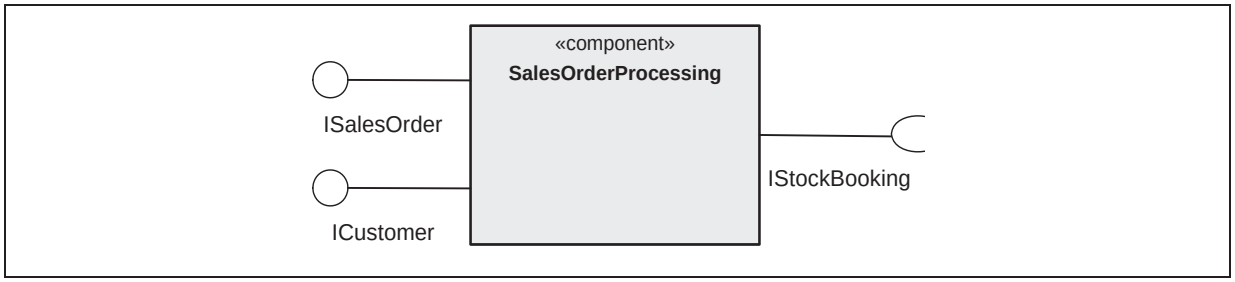
As a specification must comprise different aspects it is frequently divided into different specification levels. Objects to be specified include business terms, business tasks, interface signatures, behavioral and coordination constraints, non-functional attributes as well as general and commercial information. This paper focuses on the technical levels which are therefore discussed now in a bit more detail: [BJPW99], [Turo02] and [Over04] largely agree on its specification extent and use levels for signatures, behavior and coordination. The signature (or syntactic) level consists of signature lists which include definitions for types, constants, operations, exceptions and events. Frequently used notations on this level are OMG IDL [OMG02] and UML interface diagrams [OMG05c]. Agreements at behavioral level describe how the component acts in general and in borderline cases. This is achieved by defining constraints (pre- and postconditions) based on the idea of designing applications by contract [Meye92]. Most current specification approaches use the *UML Object Constraint Language* (OCL) [OMG05a] to specify behavioral aspects. Agreements at coordination level regulate the sequence in which component services may be invoked. Possible notations for this level include finite state machines [OMG05c], temporal operators [CoTu01] or Petri-Nets [Petr62].

Interface and behavioral specifications are now illustrated by a simplified exemplary component *SalesOrderProcessing* which will be used as example throughout the paper. The business task of the component is to manage sales orders and customer data. The component provides two interfaces *ISalesOrder* and *ICustomer* and has a required interface *IStockBooking*. Using a UML component diagram [OMG05c] an overview of the component and its interfaces is given in Figure 1.

Using again UML the component interfaces are specified in detail in Figure 2. The interface *ISalesOrder* features operations to create, check and cancel sales orders. Similarly the interface *ICustomer* enables to create, change and retrieve customer data. In order to decide if a sales order can be accepted, the component needs product stock information from another component which is accessed via the required interface *IStockBooking*. All interface specifications refer to data types which are also part of Figure 2.

In addition to the interface signatures, a component specification must describe the components behavior. This is achieved by defining constraints (pre- and postconditions) using UML OCL. An exemplary constraint is depicted in Figure 3: First the context of the constraint is defined – in our case the constraint is valid for the interface operation *ISalesOrder.check*. The keyword *post* indicates that the constraint is a postcondition. The constraint expression guarantees that after performing the operation the status of the sales order is either accepted or rejected.

Figure 1: Component SalesOrderProcessing



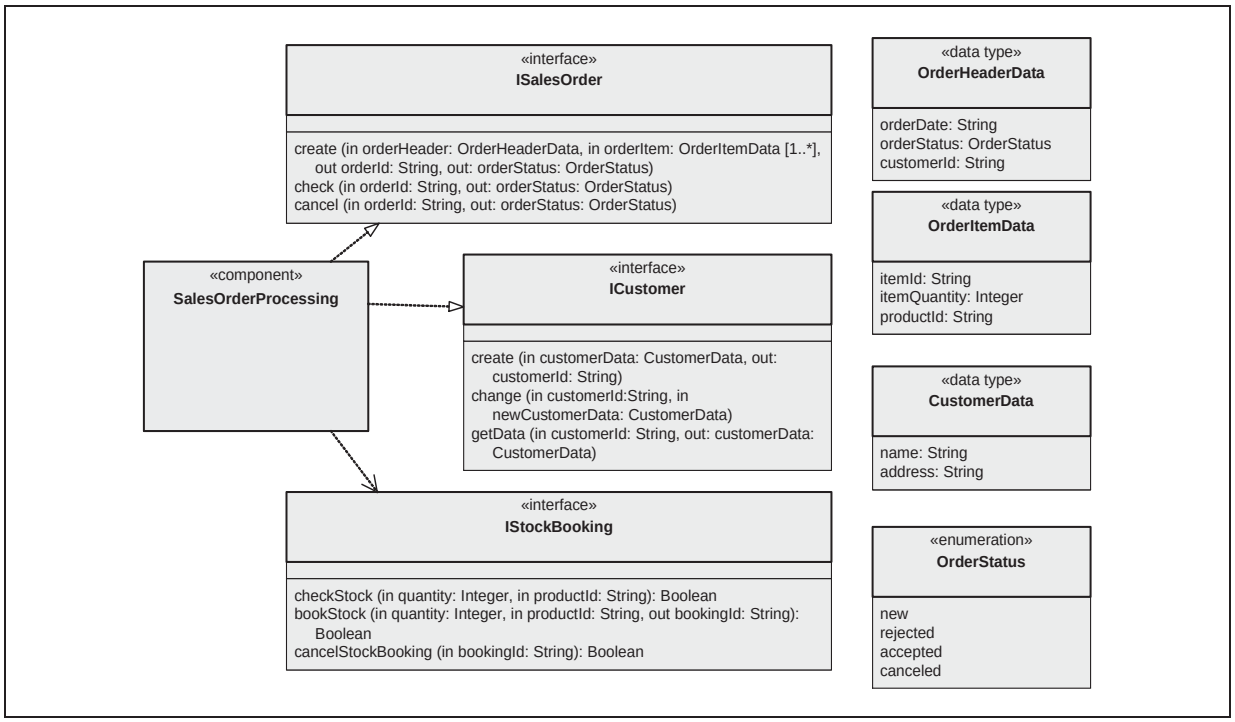


Figure 2: Interface specification for component SalesOrderProcessing

3 Black-Box Reuse and Contracts

Reusing a software component black-box means that information about the component is only available from its interfaces and its specification. In difference to that are situations where the component implementation can be studied (glass-box) or even manipulated (white-box) [SzGM02]. In the latter cases it frequently occurs that component users rely on internal implementation details – if the component provider changes such details the client will break. Therefore most authors recommend reusing software components in a black-box fashion ([McIl68], [Bosc97], [DSWi98], [SzGM02]).

The component specification, which serves as a contract between component provider and component user, is aimed at providing all necessary information about a black-box component. Such a contract specifies the components functionality and must include all information necessary for using the component correctly – any information not included is not guaranteed and considered an internal implementation detail. A component user is obliged to rely on the contract only and must not use information outside of it.

In return, the component provider can only make changes that do not invalidate the contract. In this way the component specification regulates the relationship between provider and user of a component and protects both sides from undesired expectations.

Here the issue arises, if information about the components data structure shall be part of the contract or not. Such information is often misused in white-box reuse – resulting in sacrificing the benefits of encapsulation. This has motivated the frequent conclusion not to reveal any information about the data structure. However, in our opinion this is not a coercive conclusion. Instead one must distinguish between information necessary for a component user and information not intended for a user. We will elaborate on this by discussing some examples that are based on the exemplary component introduced in Sect. 2:

- When creating a new sales order via operation *ISalesOrder.create*, it is necessary to provide a customer id. The customer id for this sales order can be retrieved at a later time by using operation *ISalesOrder.getData*. An obvious and legitimate ex-

Figure 3: Postcondition for operation ISalesOrder.check

```
context ISalesOrder::check(orderId: string, orderStatus: OrderStatus)
 post: orderStatus = OrderStatus::accepted or orderStatus = OrderStatus::rejected
```

pectation of a component user would be that both customer ids coincide (if no changes were made). Therefore this correspondence should be part of the contract.

- Also relevant for a component user is the multiplicity of the relationship between customers and sales orders. Suppose that the component allows each customer to have at most 10 sales orders (for whatever reason). If a client is not aware of this restriction he will break at the eleventh sales order of a customer.
- The component might assign sales order ids for newly created sales orders in a chronological way. This is a piece of information which is not necessary for a component user and can therefore be considered an implementation detail. If a client nevertheless relies on it (for instance by ordering sales orders based on id) and the component provider changes the algorithm (e.g. to number ranges based on order type) the client will break by its own fault. Note that this example shows the following: it is possible to obtain implementation details by mere observation without insight into the implementation!

These examples show that the correct reuse of a software component requires the contract to contain the right amount of information. If the contract includes too little information (e.g. information of first example is missing), a component user might either take the contract as final (and concludes that the component is useless) or is forced to make assumptions outside the contract. In the latter case it is very likely that the user makes additional assumptions that are not justified. If the contract includes too much information (e.g. information of third example is provided) the component provider unnecessarily restricts his ability to maintain and enhance the component.

To summarize, the component specification forms a contract between provider and user of a component and must provide all contract relevant information. A piece of information is called *contract relevant*, if it is necessary for a correct use of the component and if it is (as a consequence) explicitly guaranteed by the producer. Often it is necessary to include some information about the components data structure into the contract – this should be an abstraction of the internal data structure which includes only these properties that are relevant for the contract. Providing information in this way supports black-box reuse and does not contradict it.

4 Component Specification Data Model

In this section we demonstrate why behavioral specifications of black-box components should be equipped with a conceptual data model. We start with discussing the limitations in the current state of behavioral specifications using OCL. After that we introduce the concept of component specification data models and show how they make behavioral specifications simpler and more expressive.

4.1 Current limitations for behavioral specification of black-box components

Figure 3 showed an exemplary postcondition for the operation *ISalesOrder.check*. Besides that the operation and the interface might have the following additional constraints:

- A: Existence of a specific sales order instance required: *ISalesOrder.check* requires that the sales order with id *orderId* exists within the component. Typically this means the sales order was created earlier by operation *ISalesOrder.create* – alternatively the sales order could have been transferred to the component by initial data transfer.
- B: Dependency on status of sales order: The operation *ISalesOrder.check* can only be performed if the sales order (with id *orderId*) is in status *new*.
- C: Invariant on sales order valid for all operations: The order id plays the role of a semantic key for a sales order – that is all sales orders have a unique order id. This constraint is a requirement for operation *ISalesOrder.check* - without it the sales order to be checked might be ambiguous.

These constraints, however, *can not be expressed* adequately with OCL if only the interface specification (as shown in Figure 2) is available. To understand this we need to look closer at the component. Business components often manage business data and store them persistently. Our component *SalesOrderProcessing* manages for instance sales orders - it allows creating a sales order which is stored within the component and allows checking or canceling this sales order at a later time. (The collection of data stored by a component at a given time is sometimes referred to as the internal state of the component [ChDa01].) The constraints A - C express conditions about such business data – constraint A for example demands that the sales order with id *orderId* is known to the component. The business data managed by the component is, however, not

represented in the interface specification in Figure 2. Note that OCL constraints always refer to a UML model and can only use elements of this model. Therefore OCL constraints based on Figure 2 can only use elements of the interface definitions (as operations and parameters) and can not refer to the business data stored by the component.

To summarize: Some behavioral constraints can not be expressed using interface specifications alone because interface specifications contain no representation of the business data managed by the component. As a consequence the behavior of business components can *not* be specified *completely* without additional means.

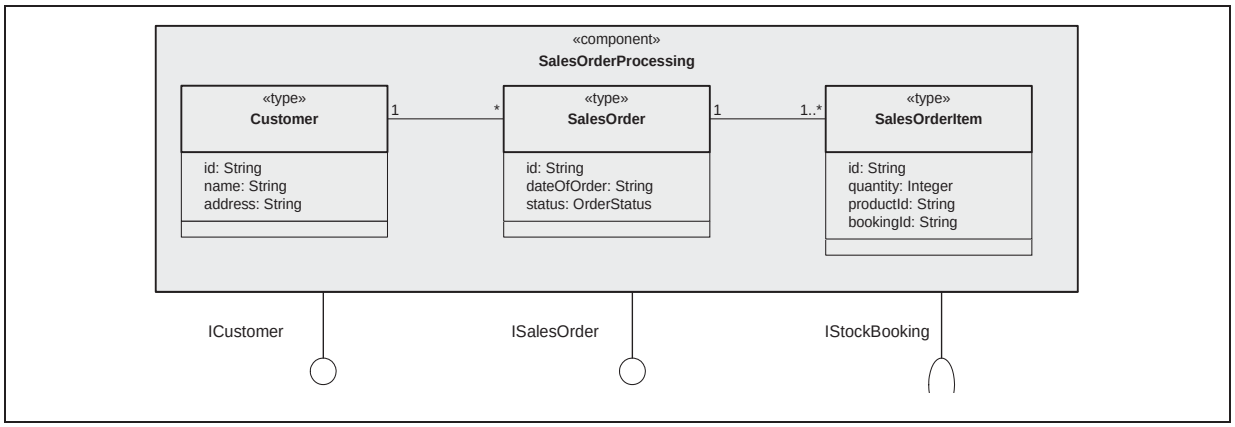
4.2 Solution: defining a component specification data model

To overcome these limitations one must include the business data into the component specification. For this purpose we introduce the concept of a *component specification data model* – a specific model, which is part of a component specification and which aim is to represent the business data managed by the component on a logical level. Such a model must only contain contract relevant information and does not need to correspond to the actual implementation. The model is an abstraction of the internal data structure which omits internal details and structures the data as it is seen from the outside.

A component specification data model is realized in the following way:

- In UML 2.0 the model element *component* can have an internal view. We use this internal view to represent the business data.
- Each contract relevant business entity type is represented by a type and stands for a number of business entities. The UML metamodel element *type* is a class with the standard stereotype «type» and is introduced for such purposes: The stereotype «type» is used to specify "a domain of objects … without defining the physical implementation of those objects." [OMG05c].
- Contract relevant properties of a business entity are described by attributes of a type and relationships between business entities are represented by associations.
- Component services are only provided by the component interfaces which are described by the interface specification. The assignment of operations to the types in the specification data model is therefore superfluous and will potentially reveal internal implementation details. (Note that in general UML types are allowed to have operations.)
- How business data is exactly represented as types, attributes and associations is a modeling decision of the person who specifies the component. Important is to include only information that shall become part of the components contract. The structure of the data model should be oriented towards the data structure in the component interfaces. In this way the specification data model will have a structure that is closer to the structure of the external visible data (instead of the internal realization), will be easier understandable from the outside and will make the resulting constraints simpler and more intuitive.
- As the specification data model is intended to enhance OCL expressiveness and to simplify OCL constraints, it is possible to include additional or redundant features into the model if this yields easier constraint definitions. For example: If the model con-

Figure 4: Component specification data model for component SalesOrderProcessing



```
context SalesOrder
 inv: SalesOrder.allInstances()->forAll(i1, i2 | i1 <> i2 implies i1.id <> i2.id)
```

Figure 5: Invariant for type SalesOrder

tains a complex navigation path from type A to type B and this navigation is needed in several constraints, one can define a new attribute for type A representing the navigation result. Within constraints one can now extract this feature directly from type A making the constraints much easier to write and to understand. This is in accordance with the style guidelines recommended in the standard book on OCL [WaKl99].

A component specification data model for the exemplary component *SalesOrderProcessing* is depicted in Figure 4. The model contains the types *Customer*, *SalesOrder* and *SalesOrderItem* and thus shows that the component manages an arbitrary number of sales orders, sales order items and customers. The attributes of the type *Customer* expose that the component stores information about customer id, name and address. Similarly the attributes of the type *SalesOrder* show that the component knows for each sales order its id, date of order and status. The association between the types *Customer* and *SalesOrder* reveals that each sales order is connected to exactly one customer and that each customer can request many sales orders. Beside the association one could enhance the type *SalesOrder* by an additional derived attribute */customerId* representing the id of the associated customer. We forgo this extension, however, as the simplification would only be minimal in this case.

Using the component specification data model from Figure 4 we can now provide a precise behavioral specification of the component. First we specify in Figure 5 an invariant. This invariant is valid for the type *SalesOrder* and thus independent from specific operations. It guarantees that different sales orders always differ in the value of their id – therefore the attribute *id* is a semantic key for sales orders (compare constraint C above).

In Figure 6 we show the behavioral specification of the interface operation *ISalesOrder.check*. This operation performs the checking of a specific sales order. The sales order in question is identified by the input parameter *orderId* and the result is returned in the output parameter *orderStatus*. The first precondition demands that the operation can only be performed for existing sales orders – that is there must exist a sales order instance which *id* equals the value of *orderId* (compare constraint A above). Note that the invariant in Figure 5 guarantees that there is at most one sales order fulfilling the condition. The second precondition demands that the sales order must have status *new* (compare constraint B stated earlier).

The postconditions assure that after the operation call the manipulated sales order instance is either in status *accepted* or in status *rejected* and its status is returned in the parameter *orderStatus*. Note that these postconditions are stronger than the constraint formulated in Figure 3: Although one might reasonably expect from Figure 3 that the status returned in the parameter *orderStatus* corresponds to the status of the sales order instance, this is not explicitly specified.

4.3 Discussion of the solution

This section discusses the advantages of the solution presented in Sect. 4.2 and explains why it does not contradict the black-box paradigm. Additionally we discuss alternative approaches (finite state machines, temporal operators) and explain why they can not solve the problem at hand.

Using a component specification data model allows to represent the business data managed by a component directly in the specification. This provides the following advantages: Firstly, one can express invariants that constrain the business data managed by the component which avoids repeating such constraints in pre- and postconditions of several operations. Secondly, it becomes possible to request the existence of business entities that are stored in the component. Thirdly, one can express pre- and postconditions that concern data which is not part of an operation interface (as parameters in configuration files). To summarize, only the use of a component specification data model allows specifying the components behavior *completely* because it enables to express the relationship between operations and business data. As a side effect using a data model also simplifies behavioral specifications.

A component specification data model is a means to include all relevant information into the component contract and shall contain only contract relevant information. In this way the specification data model supports black-box reuse and does not contradict it.

Alternatively to our solution some authors (e.g. [Turo02]) employ coordination level techniques to describe changes in business data. To express that a sales order must be in status *new* one can state (using for instance temporal operators [CoTu01])

```
context ISalesOrder::check(orderId: string, orderStatus: OrderStatus)
 pre: SalesOrder.allInstances()->exists(id = orderId)
 pre: SalesOrder.allInstances()->select(id = orderId).status = OrderStatus::new
 post: let ord1: SalesOrder = SalesOrder.allInstances()->select(id = orderId) in
       ord1.status = OrderStatus::accepted or ord1.status = OrderStatus::rejected
 post: SalesOrder.allInstances()->select(id = orderId).status = orderStatus
```

Figure 6: Constraints for operation ISalesOrder.check

that the sales order must have been created earlier (by operation *create*) and that it must not have been changed afterwards by operations *check* or *cancel*. The disadvantages of this approach are: Firstly, it requires that all data needed for a constraint is available in the operation signature and that all changes to business data are always done via the interface. These assumptions are not always fulfilled [Acke01] - business data might for example have come via initial data transfer to the component. Secondly, the resulting constraints are complicated and not easily understandable. Therefore the use of temporal operators does not solve the problem identified earlier.

One might think that finite state machines [OMG05c] are an alternative to a specification data model. Finite state machines are used by some authors to express constraints on coordination level [Over04]. For our example in Figure 2 one could e.g. define a finite state machine for the attribute *status* of *SalesOrder* (with the states *new, rejected*, *accepted* and *canceled*) and specify that operation *ISalesOrder.check* can only be executed in state *new*. Employing finite state machines is a useful *addition* in component specifications but can not replace a specification data model: Listing alone all possible states for a string attribute is virtually impossible. Moreover, a data model in practice might have many attributes and each of them many possible instances. For a state machine approach one could either define one state machine for the component using multi-dimension states (one dimension for each attribute) or define one state-machine for each attribute and specify numerous constraints between these machines. Both approaches are not feasible and there are no clear advantages compared to a specification data model.

5 Interface Specification Data Models

In the last section we introduced the idea of a component specification data model which represents the business data managed by the component. Such a model contains all the contract relevant data manipulated by all interfaces of the component. However, for component specifications it might not be enough to have a conceptual data model only on the component level. A component consumer might also be interested in the data manipulated by one interface, for instance if he does not plan to use all interfaces.

[ChDa01] introduce Interface Information Models (IIMs) that represent the state of the component on which an interface depends. In their approach they define information models only on interface level (separately for each interface) and thus do not have a model valid for the whole component. The disadvantages of this solution are the missing “big picture” and the necessity to finally tie the loose ends together. For the latter they are forced to introduce additional inter-interface constraints relating the separate information models.

We have argued that component specifications need conceptual data models both on component and interface level and that those models should be closely related. In this section we develop a solution to this requirement.

A conceptual data model for an interface shall represent all business data of the component that is manipulated by that interface. We can observe that one interface manipulates possibly less, but never more business data than all interfaces together. Moreover, the structure of the business data manipulated by the interface can not be different from the structure of all business data. Therefore it is possible to define the conceptual data models on interface level as projections of the model on component level. Following this idea we define an *interface specification data model* as subset of the component specification data model containing all business data that is manipulated by the interface. Note that we have chosen the name *interface specification data model* instead of *interface information model* [ChDa01] for two reasons: First, the terms *specification* and *data* stress that the issue is a data model only for specification purposes and are therefore better suited for black-box component specification than the rather general term *information*. Second, because of their close relationship we want to name the models on component and interface level similarly, but the term *component information model* is already used in other contexts [Brow98].

To define the specification data model for an interface, it is necessary to specify which elements (types, attributes, associations) of the component

specification data model are accessed by the interface. In the UML model this is technically realized by the "Uses" construct: For each (component specification data) model element used by an interface a "uses" relation is defined from the interface to the element. In a tool used in praxis it will be better to present this relation on some detail screen for the elements – to show each relation as line in the model would overcrowd the model and substantially lower its clarity.

Figure 7 shows the component specification data model from Figure 4 enhanced by the information which interface uses which model elements. For reasons of clarity we use a workaround to display the uses-relations in the diagram: Each interface was assigned a unique number that is displayed as tagged value. As an example consider interface *ISalesOrder* that carries the tagged value {2}. Additionally each element of the specification data model shows the numbers of those interfaces it is related to. The type *Customer* for instance carries the tagged value {1,2} and is therefore used by the interfaces *ICustomer* ({1}) and *ISalesOrder* ({2}).

Utilizing the component specification data model and the defined uses-relations it is simple to derive an interface specification data model. Figure 8 shows the data model for the interface *ICustomer*. The data model contains only the type *Customer* and all its attributes.

The interface specification data model for the interface *ISalesOrder* is displayed in Figure 9. The interface manipulates sales orders and their items and therefore its data model contains the types *SalesOrder* and *SalesOrderItem*, all their attributes and the association between them. When creating a new sales order a relation to the requesting customer is created. Therefore the type *Customer*, its attribute *id* and its association to *SalesOrder* are also part of the Figure 9 – the other attributes of *Customer*, however, are not accessed.

For reasons of completeness Figure 9 also contains the required interface *IStockBooking* because it is a requirement for the correct functioning of the interface *ISalesOrder*. (Note that in difference *ICustomer* does not need *IStockBooking*.) The relationship to *IStockBooking* is, however, not kept by the specification data model and therefore here of no further interest.

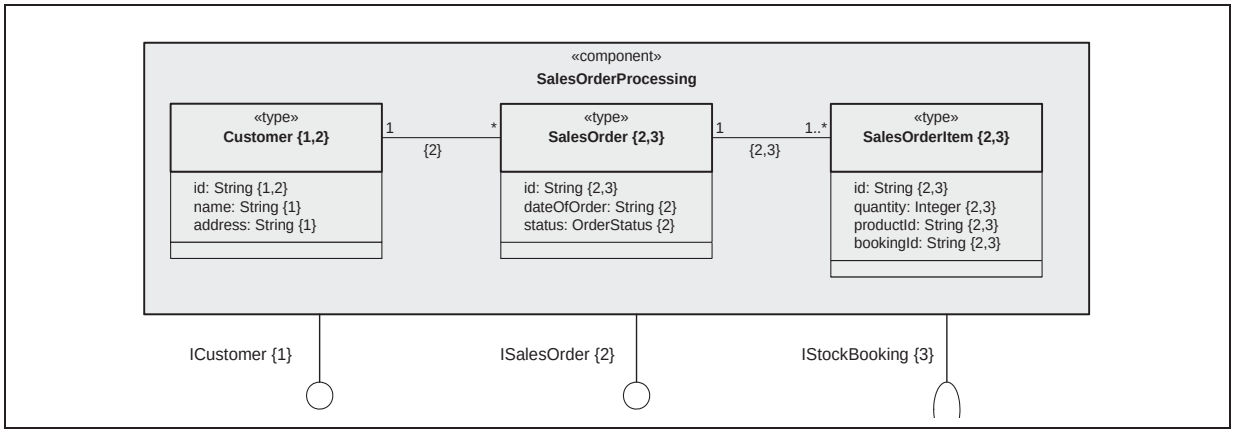
To ensure that the resulting interface specification data models are syntactically correct, it is necessary to impose certain constraints. For instance an attribute can only be part of an interface specification data model if its owning type is also included and an association requires both related types to be present. Those constraints are formulated as OCL constraints on the level of the UML metamodel (level M2 in the four-layer metamodel hierarchy of UML [OMG05b]) and must be fulfilled when constructing an interface specification data model. In practice those constraints will be checked by a specification tool.

6 Related Work

Although component specifications have already been addressed by various authors, there are only few approaches ([BJPW99], [Turo02], [Over04]) towards a comprehensive specification of black-box components (see Sect. 2).

All of these comprehensive approaches identify the need for behavioral specifications and propose to use pre- and postconditions based on UML OCL [OMG05a]. All of them neither use an (explicit) specification data model nor discuss the issue of employing one nor give any detailed account how pre- and postconditions shall be formed. To under-

Figure 7: Component specification data model for component SalesOrderProcessing including dependency on interfaces



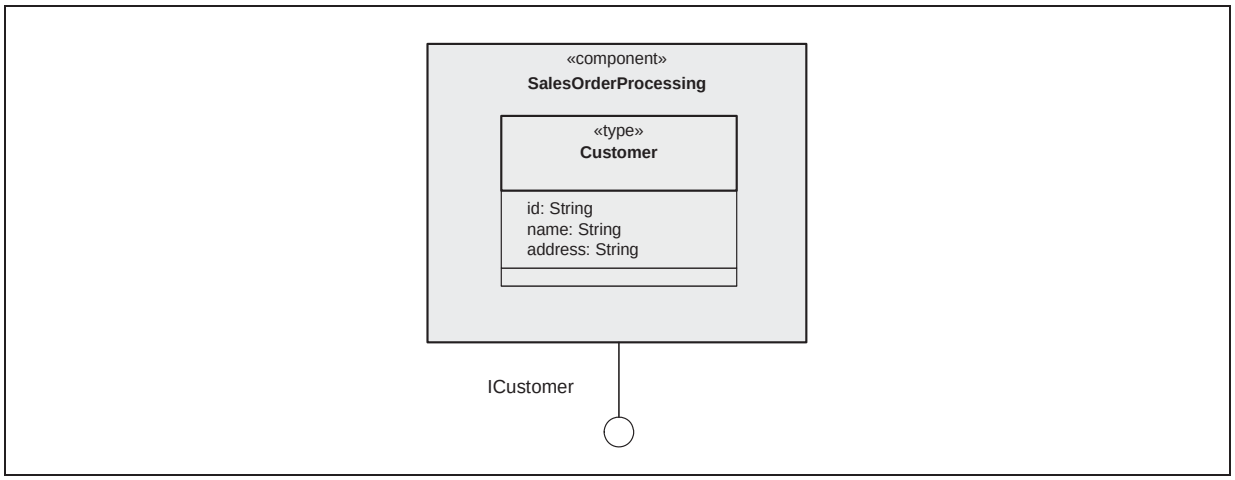
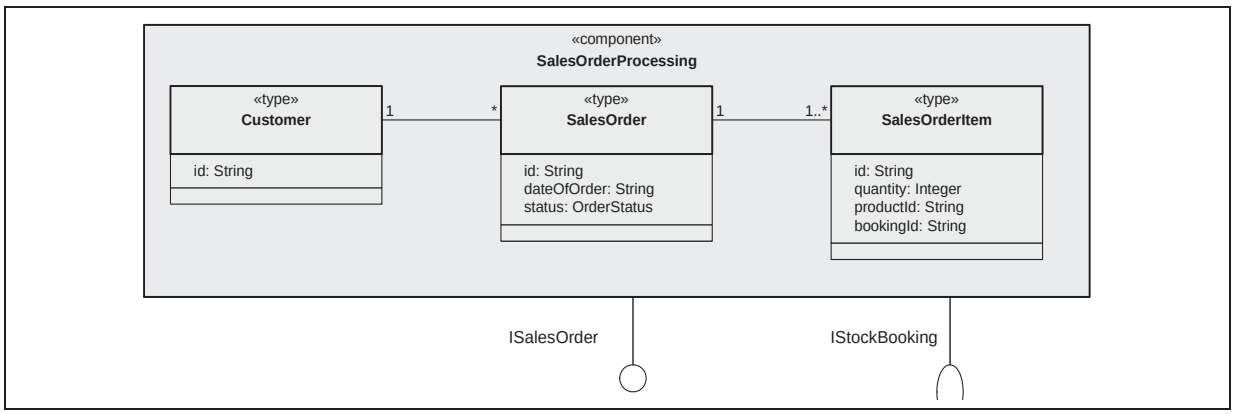


Figure 8: Interface specification data model for interface ICustomer

stand their method of behavioral specification we can only resort to analyzing the given examples. [Turo02] and [Over04] refer in their examples to the business data of the component (for instance a pre-condition demands that a certain account exists) using interface specification elements alone. While the intention is right, its realization via interface constructs is technically not correct. Resolving the problem requires a specification data model. The examples in [BJPW99] use business data of the component in a technically correct way. One must note, however, that the examples in the given form only work under very strong prerequisites: all data needed in constraints is available via query operations, the component manages only one business entity type and for each business entity instance a new component instance is invoked. As these conditions can not be assumed in all situations the behavioral specification method of [BJPW99] does not present a general solution. We only know about our own publications ([AcTu03], [Acke05], [AcTu06]) that employ a specification data model for black-box specification. In those works such a model is employed, but its application is neither justified nor sufficiently explained.

Closest to our approach is the work of [ChDa01] - they explicitly discuss the need to represent business data of a component in a model and introduce the idea of interface information models. Our work presents an extension of their concept in two ways: Firstly, by introducing information models on component level from which models on interface level can be deduced we overcome their shortcomings of a missing “big picture” and of having separate models that need to be integrated by additional inter-interface constraints. Secondly, we extended their idea to *black-box* component specification and have argued that this approach is indeed justified. Regarding the black-box issue note the following: [ChDa01] presents a component-based development process. Due to their top-down approach they focus on component identification, design and implementation but not on component reuse and discovery. Consequently they use component specifications to denote requirements and to guide component im-

Figure 9: Interface specification data model for interface ISalesOrder



plementation and do not consider explicitly component discovery. Their process does not yield a specification document containing all externally visible component properties (and only these). Therefore this specification approach is not sufficient for component discovery by third parties and can not be considered as a black-box component specification approach.

An approach similar to [ChDa01] is presented in [DSWi98] which uses so called type models to specify business data. The difference to our approach is their focus on specification as prerequisite to implementation (again no black-box specification approach) and on the use of object-oriented implementations across component boundaries. Note that the latter might not always be the case for reusable black-box components [SzGM02].

[Meye92] introduced "design by contract" for object-oriented classes by specifying constraints (invariants, pre- and postconditions) which the attributes and operations of a class must adhere to. When specifying behavioral aspects of black-box software components the ideas of Meyer cannot be directly employed – doing so would reveal internal details of the implementation. His ideas were instead transformed and applied to the components interfaces which are specified by pre- and postconditions. In this way, however, only half of the concepts from "design by contract" are used for components: invariants as well as constraints regarding the business data of a component cannot be expressed (for details compare Sect. 4.1). The introduction of a specification data model closes this gap and in this way completes the transformation of "design by contract" principles to component specifications.

7 Summary

This paper discussed in detail the use of conceptual data models in black-box component specification: We showed why current specification approaches are inadequate and introduced the concept of component specification data models to overcome these limitations. Note that component specification data models shall contain contract relevant data only and do not contain implementation details – this is in accordance with the black-box component paradigm. Furthermore we have argued that component specifications need conceptual data models both on component and interface level and that those models should be closely related. To fulfill this requirement we introduced interface specification data models that are derived from the component specification data model.

References

[Acke01] Ackermann, J.: Fallstudie zur Spezifikation von Fachkomponenten. In: Turowski, K. (ed.): 2. Workshop Modellierung und Spezifikation von Fachkomponenten. Bamberg 2001, pp. 1-66. (In German)

[Acke05] Ackermann, J.: Frequently Occurring Patterns in Behavioral Specification of Software Components. In: Turowski, K.; Zaha, J.M. (eds.): Component-Oriented Enterprise Applications - Proceedings of the Conference on Component-Oriented Enterprise Applications (COEA 2005). LNI P-70. Erfurt 2005, pp. 41-56.

[AcTu03] Ackermann, J.; Turowski, K.: Specification of Customizable Business Components. In: Chroust, G.; Hofer, S. (eds.): Euromicro Conference 2003. Belek-Antalya 2003, pp. 391-394.

[AcTu06] Ackermann, J., Turowski, K.: A Library of OCL Specification Patterns for Behavioral Specification of Software Components. In: Dubois, E.; Pohl, K. (eds.): CAiSE 2006. LNCS 4001. Springer-Verlag, Berlin Heidelberg 2006, pp. 255-269.

[BJPW99] Beugnard, A.; Jézéquel, J.-M.; Plouzeau, N.; Watkins, D.: Making Components Contract Aware. In: IEEE Computer 32 (1999) 7, pp. 38-44.

[Bosc97] Bosch, J.: Adapting Object-Oriented Components. In: Proceedings of the 2nd International Workshop on Component-Oriented Programming (WCOP '97). Turku 1997, pp. 13-21.

[Brow00] Brown, A.W.: Large-Scale Component-Based Development. Prentice Hall, Upper Saddle River 2000.

[Brow98] Brown, A.W.: From Component Infrastructure to Component-Based Development. In: Position Paper of the 1998 International Workshop on Component-Based Software Engineering. Tokyo 1998.

[ChDa01] Cheesman, J.; Daniels, J.: UML Components. Addison-Wesley, Boston 2001.

[CoTu01] Conrad, S.; Turowski, K.: Temporal OCL: Meeting Specification Demands for Business Components. In: Siau, K.; Halpin, T. (eds.): Unified Modeling Language: Systems Analysis, Design and Development Issues. Idea Group, Hershey 2001, pp. 151-165.

[DSWi98] D'Souza, D.F.; Wills, A.C.: Objects, Components, and Frameworks with UML: The Catalysis Approach. Addison-Wesley, Reading 1998.

[GeGh06] Geisterfer, C. J. M.; Ghosh, S.: Software Component Specification: A Study in Perspective of Component Selection and Reuse. In: Proceedings of the 5th International Conference on COTS Based Software Systems (ICCBSS). Orlando (USA) 2006, pp.100-108.

[HaTu02] Hahn, H.; Turowski, K.: General Existence of Component Markets. In: Trappl, R. (ed.): Sixteenth European Meeting on Cybernetics and Systems Research (EMCSR). Vol. 1, Vienna 2002, pp. 105-110.

[HeLi01] Hemer, D.; Lindsay, P.: Specification-based retrieval strategies for module reuse. In: Grant, D.; Sterling, L. (eds.): Proceedings 2001 Australian Software Engineering Conference. IEEE Computer Society. Canberra 2001, pp. 235-243.

[McIl68] McIlroy, M. D.: Mass Produced Software Components. In: Naur, P.; Randell, B. (eds.): Software Engineering: Report on a Conference by the NATO Science Committee. Brussels 1968, pp. 138-150.

[Meye92] Meyer, B.: Applying "Design by Contract". In: IEEE Computer 10 (1992), pp. 40-51.

[OMG02] OMG (ed.): CORBA Components. OMG Specification, Version 3.0 02-06-65. Framingham 2002.

[OMG05a] OMG (ed.): Object Constraint Language. Version 2.0, formal/06-05-01. http://www.omg.org/technology/documents, (2007-02-02).

[OMG05b] OMG (ed.): Unified Modeling Language: Infrastructure. Version 2.0, formal/05-07-05. http://www.omg.org/technology/documents, (2007-02-02).

[OMG05c] OMG (ed.): Unified Modeling Language: Superstructure. Version 2.0, formal/05-07-04. http://www.omg.org/technology/documents, (2007-02-02).

[Over04] Overhage, S.: UnSCom: A Standardized Framework for the Specification of Software Components. In: Weske, M.; Liggesmeyer, P. (ed.): Object-Oriented and Internet-Based Technologies, Proceedings of the 5th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World (NOD 2004). Springer LNCS 3263. Erfurt 2004, pp. 169-184.

[Petr62] Petri, C. A.: Fundamentals of a Theory of Asynchronous Information Flow. In: Information Processing 62. IFIP 1962, pp. 386–391.

[SzGM02] Szyperski, C.; Gruntz, D.; Murer, S.: Component Software: Beyond Object-Oriented Programming. 2. ed. Addison-Wesley, Harlow 2002.

[Turo02] Turowski, K. (ed.): Standardized Specification of Business Components: Memorandum of the working group 5.10.3 Component Oriented Business Application System. Augsburg 2002. http://www.fachkomponenten.de, (2007-02-02)

[WaKl99] Warmer, J.; Klepper, A.: The Object Constraint Language: Precise Modeling with UML. Addison-Wesley, Reading 1999.

[YeSt97] Yellin, D.; Strom, R.: Protocol Specifications and Component Adaptors. In: ACM Transactions on Programming Languages and Systems 19 (1997), pp. 292–333.

Jörg Ackermann

Chair of Business Informatics and Systems Engineering
University of Augsburg
Universitätsstraße 16
86135 Augsburg
Germany
joerg.ackermann@wiwi.uni-augsburg.de