A Library of OCL Specification Patterns for Behavioral Specification of Software Components

Jörg Ackermann and Klaus Turowski

Chair of Business Informatics and Systems Engineering,
University of Augsburg, Universitätsstr. 16, 86135 Augsburg, Germany
{joerg.ackermann, klaus.turowski}@wiwi.uni-augsburg.de

Abstract. One important aspect in building trusted information systems is the precise specification of systems and system parts. This applies even more for information systems built from COTS components. To specify behavioral aspects of software components the UML Object Constraint Language (OCL) is well suited. One current problem in component specifications comes from the fact that editing OCL constraints manually is time consuming and error-prone. To simplify constraint definition we propose to use specification patterns for which OCL constraints can be generated automatically. In this paper we outline this solution proposal and present a library of reusable OCL specification patterns.

Keywords: Component-Based Information Systems, Software Component Specification, OCL Specification Patterns.

1 Introduction

One important aspect in building trusted information systems is the precise specification of systems and system parts: the specification can help to close the (frequently be observable) gap between the expected and the actual behavior of a system. Specifications become even more important if information systems are built from COTS components acquired from a variety of vendors [23]. A precise and reliable specification of COTS components supports sound selection and trust in its correct functioning [10]. Moreover, component specifications are a prerequisite for a composition methodology and tool support [20]. For these reasons the specification of software components is of utmost importance for building trusted information systems out of components.

To specify the behavioral aspects of software components the *UML Object Constraint Language* (OCL) [18] is frequently employed (see Sect. 2). Using a formal language like OCL, however, causes one of the current problems in component specifications: Editing OCL constraints manually is time consuming and error-prone (see Sect. 3). To simplify constraint definition we propose to utilize specification patterns for which OCL constraints can be generated automatically (see Sect. 4). For that we present a library of OCL specification patterns (Sect. 5) and discuss a technique to describe and formally specify them (Sect. 6). We conclude with discussion of related work (Sect. 7) and a summary (Sect. 8). The contribution of the project is the simplification of component specification by utilizing reusable OCL specifications. The results were developed for software components but are general enough to be interesting for anyone using OCL to specify information systems.

2 Behavioral Specification of Software Components

The appropriate and standardized specification of 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 a component provides under which conditions [22]. Various authors addressed specifications for specific tasks of the development process as e.g. design and implementation [8,9], component adaptation [24] or component selection [14]. Approaches towards comprehensive specification of software components are few and include [7,20,22]. Objects to be specified are e.g. business terms, business tasks, interface signatures, behavior and coordination constraints and non-functional attributes.

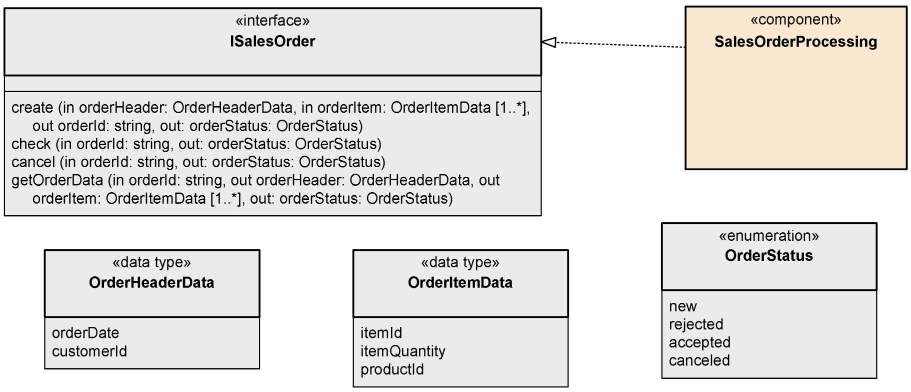

«interface»
ISalesOrder
create (in orderHeader: OrderHeaderData, in orderItem: OrderItemData [1..*], out orderId: string, out: orderStatus: OrderStatus)
check (in orderId: string, out: orderStatus: OrderStatus)
cancel (in orderId: string, out: orderStatus: OrderStatus)
getOrderData (in orderId: string, out orderHeader: OrderHeaderData, out orderItem: OrderItemData [1..*], out: orderStatus: OrderStatus)
«component»
SalesOrderProcessing
«data type»
OrderHeaderData
orderDate
customerId
«data type»
OrderItemData
itemId
itemQuantity
productId
«enumeration»
OrderStatus
new
rejected
accepted
canceled

Fig. 1. Interface specification of component *SalesOrderProcessing*

Behavioral specifications (which are topic of this paper) describe how the component behaves in general and in borderline cases. This is achieved by defining constraints (invariants, pre- and postconditions) based on the idea of designing applications by contract [16]. OCL is one of the most used techniques to express such constraints – cf. e.g. [8,9,20,22].

To illustrate how behavioral aspects of software components are specified we introduce a simplified exemplary component *SalesOrderProcessing*. The business task of the component is to manage sales orders. This component is used as example throughout the rest of the paper.

Fig. 1 shows the interface specification of *SalesOrderProcessing* using UML [19]. We see that the component offers the interface *ISalesOrder* with operations to create, check, cancel or retrieve specific sales orders. The data types needed are also defined in Fig. 1. Note that in practice the component would have additional operations and might offer additional order properties. For sake of simplicity we restricted ourselves to the simple form shown in Fig. 1 which will be sufficient as example for this paper.

To specify the information objects belonging to the component (on a logical level) one can use a specification data model which is realized as a UML type diagram and is part of the behavioral specification [5]. Fig. 2 displays such a model for the component *SalesOrderProcessing*. It shows that the component manages sales orders (with attributes id, date of order, status, customer id) and sales order items (with attributes id, quantity, product id) and that there is a one-to-many relationship between sales orders and sales order items.

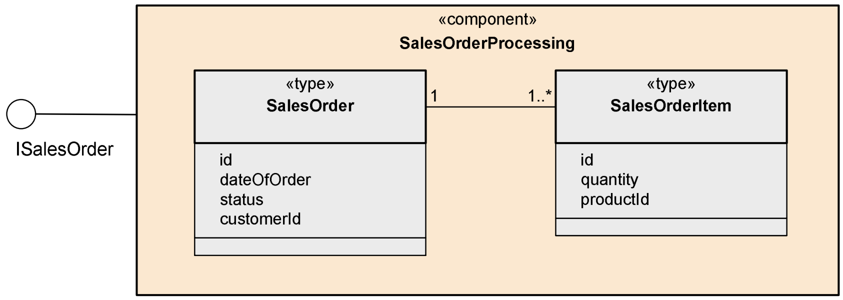


Fig. 2. Specification data model for component *SalesOrderProcessing*

Note that operation parameters for software components are usually value-based (no object-oriented instances are passed). As a consequence the coupling between interface definition (cf. Fig. 1) and specification data model (cf. Fig. 2) is only loose. The operation *ISalesOrder.check* e.g. is semantically an instance method in the sense that it manipulates one specific sales order. Technically, however, the operation is called for the component (and not a sales order instance) and the sales order to be checked is identified by the parameter *orderId*. Therefore the consequences of operation calls on the specification data model must be specified explicitly.

The behavioral specification of a component is based on its interface specification and on its specification data model and consists of OCL expressions that constrain the components operations – for an example see Fig. 3. The first constraint is an invariant for type *SalesOrder*: It guarantees that different sales orders always differ in the value of their id – that is the attribute *id* is a semantic key for sales orders. By defining an invariant this constraint needs only to be formulated once and does not need to be repeated in several pre-and postconditions. The remaining constraints in Fig. 3 concern the operation *ISalesOrder.create*: The precondition demands that the field *customerId* of parameter *orderHeader* must not contain an empty string when calling the operation. The first postcondition guarantees that an instance of class *SalesOrder* (which id equals the value of the output parameter *orderId*) was created by the operation. The second postcondition assures that the newly created sales order instance is in status *new*. Note that Fig. 3 shows only some constraints and does not contain a complete behavioral specification.

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

context ISalesOrder::create(orderHeader: OrderHeaderData, order-
Item: OrderItemData, orderId: string, orderStatus: OrderStatus)
 pre:  orderHeader.customerId <> ''
 post: let inst: SalesOrder = SalesOrder.allInstances()
              ->select(i1 | (i1.id = orderId))->any(true) in
       inst.oclIsNew
 post: let inst: SalesOrder = SalesOrder.allInstances()
              ->select(i1 | (i1.id = orderId))->any(true) in
       inst.status = OrderStatus::new
```

Fig. 3. Exemplary behavioral constraints for component *SalesOrderProcessing*

3 Problems in Behavioral Specification of Components

Most component specification approaches recommend notations in formal languages since they promise a common understanding of specification results across different developers and companies. The use of formal methods, however, is not undisputed. Some authors argue that the required effort is too high and the intelligibility of the specification results is too low – for a discussion of advantages and liabilities of formal methods compare [13].

The disadvantages of earlier formal methods are reduced by UML OCL [18]: The notation of OCL has a comparatively simple structure and is oriented towards the syntax of object-oriented programming languages. Software developers can therefore handle OCL much easier than earlier formal methods that were based on set theory and predicate logic. This is one reason why OCL is recommended by many authors for the specification of software components.

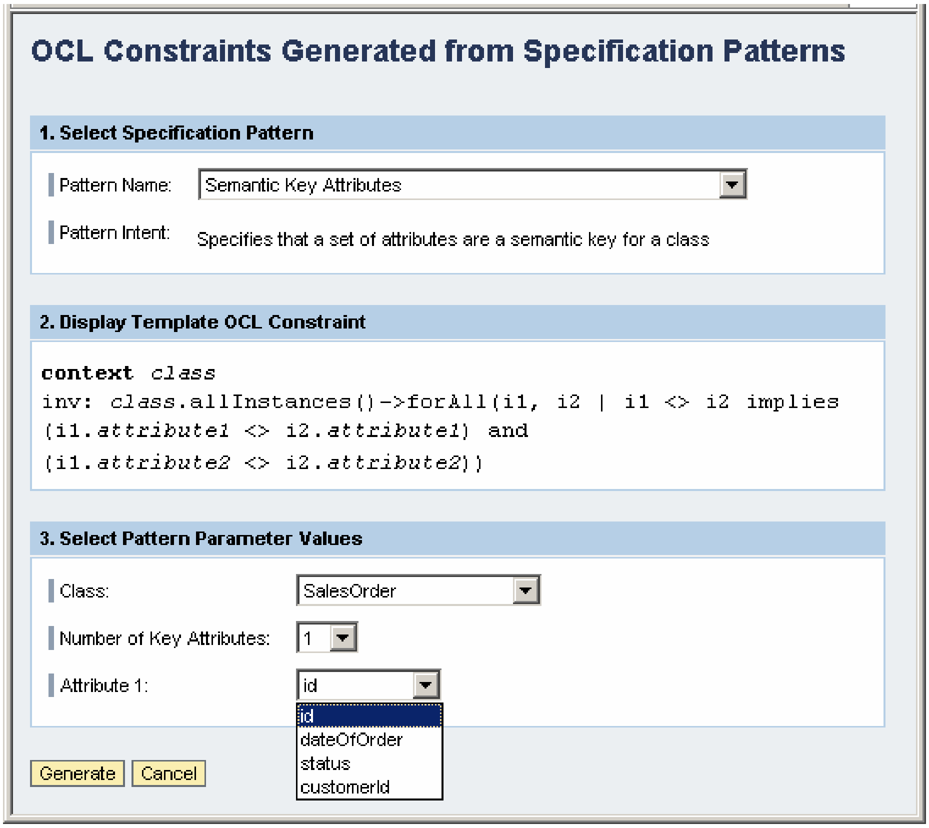
Despite its advantages OCL can not solve all problems associated with the use of formal methods: One result of two case studies specifying business components [1,2] was the insight that editing OCL constraints manually is nevertheless time consuming and error-prone. Similar experiences were made by other authors that use OCL constraints in specifications (outside the component area), e.g. [12,15]. They conclude that it takes a considerable effort to master OCL and use it effectively.

It should be noted that behavioral aspects (where OCL is used) have a great importance for component specifications: In the specification of a rather simple component in case study [2], for example, the behavioral aspects filled 57 (of altogether 81) pages and required a tremendous amount of work. For component specifications to be practical it is therefore mandatory to simplify the authoring of OCL constraints.

4 Solution Proposal: Utilizing OCL Specification Patterns

Solution strategies to simplify OCL specifications include better tool support (to reduce errors) and an automation of constraint editing (to reduce effort) – the latter can e.g. be based on use cases or on predefined specification patterns (compare Sect. 7).

To use specification patterns seems to be particularly promising for the specification of business components: When analyzing e.g. case study [2] one finds that 70% of all OCL constraints in this study can be backtracked to few frequently occurring specification patterns.



OCL Constraints Generated from Specification Patterns

1. Select Specification Pattern

Pattern Name: Semantic Key Attributes

Pattern Intent: Specifies that a set of attributes are a semantic key for a class

2. Display Template OCL Constraint

```
context class
inv: class.allInstances()->forAll(i1, i2 | i1 <> i2 implies
(i1.attribute1 <> i2.attribute1) and
(i1.attribute2 <> i2.attribute2))
```

3. Select Pattern Parameter Values

Class: SalesOrder

Number of Key Attributes: 1

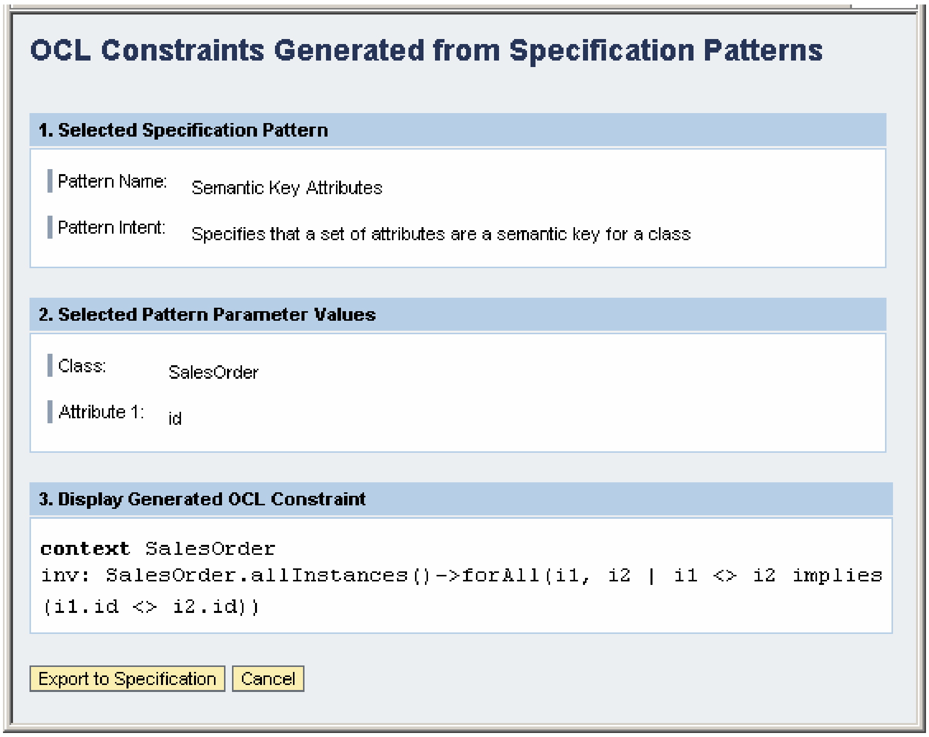
Attribute 1: id

Fig. 4. Selection screen for generating an OCL constraint

Under *(OCL) specification pattern* we understand an abstraction of OCL constraints that are similar in intention and structure but differ in the UML model elements used. Each pattern has one or more *pattern parameters* (typed by elements of the UML metamodel) that act as placeholder for the actual model elements. With *pattern instantiation* we denote a specific OCL constraint that results from binding the pattern parameters with actual UML model elements.

As an example let us consider the pattern *Semantic Key Attributes*: It represents the situation that one or more attributes of a class (in the specification data model – cf. Fig. 2) play the semantic role of a key – that is any two instances of the class differ in at least one value of the key attributes. Pattern parameters are the class and the list of key attributes. A pattern instantiation (for the class *SalesOrder* and its attribute *id*) can be seen in the upper part of Fig. 3.

If such OCL specification patterns are collected, formally described and integrated into a specification tool the specification can be simplified in the following way: Suppose the person who specifies our exemplary component is in the middle of the specification process and wants to formulate the invariant from Fig. 3. He checks the library of predefined specification patterns (which is part of his specification tool) and finds the pattern for semantic key attributes (compare section 1 of Fig. 4). After selecting this pattern the tool will show him the pattern description and an associated template OCL constraint (showing the pattern parameters in italic).


OCL Constraints Generated from Specification Patterns

1. Selected Specification Pattern

Pattern Name:	Semantic Key Attributes
Pattern Intent:	Specifies that a set of attributes are a semantic key for a class

2. Selected Pattern Parameter Values

Class:	SalesOrder
Attribute 1:	id

3. Display Generated OCL Constraint

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

Fig. 5. Display of the generated OCL constraint

The user has to select model elements for the parameters (in section 3 of Fig. 4) – in our example the class *SalesOrder* and its attribute *id* are selected. Note that the tool can be built in such a way that it restricts the input to those model elements that are allowed for a pattern – in section 3 of Fig. 4 for instance you can see that the tool only offers the attributes of class *SalesOrder* for selection.

After providing pattern and parameter values the user can start the generation. The tool checks the input for consistency and then generates the desired OCL constraint (compare section 3 of Fig. 5) which can be included into the component specification.

Applying this approach has the following advantages: For the specification provider maintenance of specifications is simplified because it becomes faster, less error-prone and requires less expert OCL knowledge. For a specification user the understanding of specifications is simplified because generated constraints are uniform and are therefore easier recognizable. Moreover, if the patterns were standardized, it would be enough to specify a pattern and the parameter values (without the generated OCL text) which would make recognition even easier.

5 A Library of OCL Specification Patterns

In this section we introduce a list of OCL specification patterns that are useful for behavioral specification of software components. To obtain this list we first studied several case studies (dealing with business components) and publications about component specifications and identified frequently occurring patterns [3]. In a second step we analyzed the preliminary list and identified additional patterns that are useful but could not be found in the first step. (Reasons to include additional patterns were for instance extending patterns to other relevant UML metamodel elements or symmetry considerations like including a constraint for deleted instances if there is one for created instances.) As result we obtained a library of altogether 18 OCL specification patterns which are subdivided into four categories. The categories and their assigned patterns are shown in Tables 1-4 – the table columns are pattern number (for easier reference), constraint type and pattern name. Note that it is conceivable that the pattern list might be extended in future.

Table 1. Specification patterns concerning only interface specifications

No.	Constraint type	Pattern name
1	Precondition	Value Specification of Input Parameter
2	Postcondition	Value Specification of Output Parameter
3	Precondition	Value Specification of Input Parameter Field
4	Postcondition	Value Specification of Output Parameter Field
5	Precondition	Value Specification of Input Table Parameter Fields
6	Postcondition	Value Specification of Output Table Parameter Fields

The behavioral specification of software components refers to the interface specification (cf. Fig. 1) and the specification data model (cf. Fig. 2). The first pattern category (cf. Table 1) contains patterns that only concern the interface specification.

Pattern 3 (*Value Specification of Input Parameter Field*) for instance is a precondition that allows restricting the value of a field of a structured input parameter of an operation. Pattern parameters are an operation, a parameter, a parameter field, one of the operators (=, <>, <, <=, > or >=) and a value (corresponding to the type of the field). Using this pattern one can e.g. demand that the field must be greater than zero or contain a specific element of an enumeration. An instantiation of this pattern is shown in the precondition of Fig. 3 which requests that the field *customerId* of input parameter *orderHeader* (of operation *ISalesOrder.create*) must not be the empty string.

Analogous preconditions can be formulated for a simple, unstructured input parameter (using pattern 1) and for a field of an input table parameter (pattern 5). Patterns 2, 4 and 6 are similar but represent postconditions assuring that an output parameter (field) has a certain value or value range.

Table 2. Specification patterns concerning only specification data models

No.	Constraint type	Pattern name
7	Invariant	Semantic Key Attributes
8	Invariant	Value Specification of Class Attribute
9	Invariant	Relationship between Class Attribute Values

The second category (cf. Table 2) features patterns that only concern the specification data model. They are independent from operation calls and thus all invariants.

An example is pattern 7 (*Semantic Key Attributes*) which was already discussed before. Pattern parameters are a class and a list of its attributes – a pattern instantiation is given by the invariant in Fig. 3. Note that the pattern is intended for one or more attributes – this is the reason for not using the operator *isUnique* which would be rather constructed for more than one attribute. Additionally it shall be mentioned that the presented patterns are rather static – they allow substituting UML model elements but do not allow structural changes. For structural variations on the pattern (e.g. the attribute *id* of class *SalesOrderItem* in Fig. 2 is only unique in the context of a specific instance of class *SalesOrder*) one has to define additional patterns.

The other patterns in Table 2 allow specifying the value of a class attribute (pattern 8) or the relationship between two attributes of the same class (pattern 9). A possible extension for the last pattern would be the relationship between two attributes of different classes which are connected by an association.

In difference to the patterns presented so far the remaining patterns (in Tables 3 and 4) address the relationship between interface specification and specification data model and are rather specific for software components (more precise: specific for value-based operation calls).

Table 3. Specification patterns concerning the existence of class instances for operation calls

No.	Constraint type	Pattern name
10	Precondition	Class Instance Existing
11	Precondition	Class Instance Not Existing
12	Postcondition	Class Instance Created
13	Postcondition	Class Instance Deleted

The third pattern category (cf. Table 3) contains patterns that deal with the existence of specific class instances for an operation call.

As an example we consider pattern 12 (*Class Instance Created*). This pattern describes a postcondition which specifies that an instance of a class (in the specification data model) was created by an operation call. The pattern parameters are the class *cl* for which an instance was created, the calling operation *op*, and two ordered sets *keyList*, *kparList* of elements identifying the class instance. The parameter *keyList* contains a list of those attributes of *cl* that form together the semantic key of *cl* (compare pattern 7). The parameter *kparList* contains those parameters or parameter fields of *op* in which the key values to identify the required instance are passed. Note that *keyList* and *kparList* must have the same number of elements and must be ordered in such a way, that corresponding entries stand at the same position within *keyList* and *kparList*. A pattern instantiation is given by the first postcondition in Fig. 3 which assures that operation *ISalesOrder.create* created a new instance of class *SalesOrder* which id equals the value of parameter *orderId*. To make the OCL constraint easier to understand first a local variable *inst* is defined for the instance in question and then the actual constraint is formulated in the context of *inst* (cf. Fig. 3).

The other patterns in Table 3 are similar (and have the same pattern parameters) as pattern 12 and can be used to request that a certain instance exists (pattern 10) or does not exist (pattern 11) before the operation call and to assure that a specific instance was deleted (pattern 13) by the operation.

Table 4. Specification patterns integrating interface specification and data model

No.	Constraint type	Pattern name
14	Precondition	Precondition on an Instance Attribute
15	Postcondition	Postcondition for an Instance Attribute
16	Postcondition	Relationship to an Instance of another Type
17	Postcondition	Equivalence of Parameter (Field) Value and Attribute Value
18	Postcondition	Equivalence of Multiline Parameter (Field) Values and Attribute Values

The fourth pattern category (cf. Table 4) collects further patterns that allow specifying prerequisites from model instances for an operation call and consequences of operation calls for model instances.

Pattern 15 (*Postcondition for an Instance Attribute*) describes a postcondition that can be used to assure that an attribute of a given class instance has a certain value (or value range) at the end of an operation call. The second postcondition in Fig. 3 e.g. states that the attribute *status* of the newly created instance of *SalesOrder* (having as id the value of *orderId*) has the value *new* at the end of calling *ISalesOrder.create*. Pattern 15 has pattern parameters for two purposes: On one hand there are the parameters *cl*, *op*, *keyList* and *kparList* which are used to identify the desired class instance (for more details compare pattern 12 as described above). On the other hand the pattern has as parameters the attribute, one of the operators (=, <>, <, <=, > or >=) and a value (corresponding to the type of the attribute) to specify the value or value range of the attribute (compare also pattern 3 described above). Looking at the postconditions in Fig. 3 one might notice that the instance variable *inst* is defined in both conditions. Behavioral specifications could be simplified if it were possible to reuse local variables like *inst* in several pre- and postconditions. UML OCL, however, only allows declaring reusable variables (via the *define* statement) for invariants and not for operation calls.

Pattern 14 is similar to pattern 15 but represents a precondition which can be used to demand that a class instance attribute has a certain value when calling an operation. Pattern 17 can be used to express that the value of a class instance attribute equals the value of a parameter (or parameter field). Such a condition is useful for create or update operations to show how class instance attributes are filled from parameters or for read operations to show how parameters are filled from instance attributes. To avoid the need to formulate many such constraints if the class has many attributes pattern 17 allows specifying the constraint for a list of attributes and parameters (or parameter fields). Pattern 18 is similar in intention but addresses operation parameters and instance properties of multiplicity greater than one. Pattern 16 is also similar and enables to specify how a parameter (or parameter field) is connected to an instance of a second class which is associated to the first class.

6 Description of OCL Specification Patterns

The specification patterns identified in Sect. 5 need to be described precisely in order to be reusable for component specifications. For that we propose to use on one hand a description scheme and on the other hand a formal specification. The formal specification is done by using OCL itself and addresses OCL experts and tool builders who wish to implement constraint generators. The description scheme addresses users of specification patterns – for this it contains all relevant specification details in a more informal way. To confront a pattern user with the full OCL specification would contradict the goal to support specifications by non-OCL experts as well. In this section we present both description approaches.

Based on the ideas of [3] we developed for pattern users a description scheme that allows displaying all relevant pattern details in a structured and uniform way. As an example Table 5 shows the description scheme for pattern number 7 (*Semantic Key Attributes*).

The first characteristic is the pattern name that identifies the pattern and serves as a short semantic explanation. The characteristic *pattern intent* contains a short statement about intention and rationale of the pattern. The characteristic *pattern parameter* lists the parameters of the pattern together with their type. Parameters can be of elementary type (like *String*) or are elements from the UML metamodel (layer M2 in the four-layer metamodel hierarchy of UML [17]). Parameters of pattern 7 are the class *cl* (of type *Class*) and the list of key attributes *keyList* (of type ordered set of *Property*). The characteristic *restrictions* denotes what conditions the pattern parameters must fulfill. In our case it is required that all elements of *keyList* are attributes of *cl*.

Table 5. Description scheme for pattern *Semantic Key Attributes*

Characteristic	**Description**
Pattern name	Semantic Key Attributes
Pattern intent	Specifies that a set of attributes are a semantic key for a class
Pattern parameter	cl: Class; keyList: Property [1..*]
Restrictions	Each element of *keyList* is an attribute of *cl*
Constraint type	Invariant
Constraint context	*cl*
Constraint body	*cl.name*.allInstances()->forAll(i1,i2 | i1 <> i2 implies (i1.*keyList[1].name* <> i2.*keyList[1].name)* and (i1.*keyList[2].name* <> i2.*keyList[2].name*))

The remaining three characteristics describe the OCL constraint the pattern represents. With *constraint type* we denote if the constraint is an invariant, pre- or postcondition. The characteristic *constraint context* stands for the OCL context of the constraint and is always one of the pattern parameters (in our example class *cl*). The final characteristic *constraint body* shows how the OCL expression to be generated looks like. For that the OCL expression is kept generic: subexpressions in normal typesetting can be left as they are (in our example e.g. `'.allInstances()'`) but subexpressions in italic stand for text that needs to be substituted by a parameter value or a parameters model name (as *cl.name* in Table 5 – because *cl* is a UML metamodel element of type *Class* we need to access its property *name* to retrieve its actual name.) Note that the chosen notation for the characteristic *constraint body* is sufficient to give an idea of the generated OCL expression but has its limitations if patterns are variable: In pattern 7 e.g. an arbitrary number of attributes can be used – in Table 5 the constraint is shown exemplary for two attributes. In this way the use of more complicated constructs (like *loop* or *iterate*) can be avoided which helps in keeping the description easily understandable. Note that the formal pattern description (as discussed below) is powerful enough to enable a precise specification.

Beside the informal pattern description each OCL specification pattern is formally specified. This formal pattern specification is necessary to avoid misunderstandings and is prerequisite for tool builders implementing constraint generators.

The basic idea how to formally describe the specification patterns is as follows: For each OCL specification pattern a specific function (called *OCL pattern function*) is defined. The pattern parameters are the input of the pattern function. Result of the pattern function is a generated OCL constraint which is returned and (if integrated with the specification tool) automatically added to the corresponding UML model element. The OCL pattern functions themselves are specified by OCL – from this specification one can determine the constraint properties (e.g. invariant) and its textual representation. All pattern functions are assigned as operations to a new class *OclPattern* which logically belongs to the layer of the UML metamodel.

As an example we consider again the pattern *Semantic Key Attributes*. For this pattern we define the OCL pattern function *Create_Inv_SemanticKeyAttributes*. Input of the function are the class *cl* and the set of attributes *keyList* – all understood as UML model elements. Result is a UML model element of type *Constraint*. The complete specification of this pattern function is shown in Fig. 6.

```
context OclPattern::Create_Inv_SemanticKeyAttributes (cl: Class,
keyList: orderedSet(Property)): Constraint
(1) pre: keyList->forAll(key | key.class = cl)

(2) post: result.oclIsNew
(3) post: result.namespace = result.context
(4) post: result.specification.isKindOf(OpaqueExpression)
(5) post: result.specification.language = 'OCL'

(6) post: result.stereotype.name = 'invariant'
(7) post: result.context = cl
(8) post: result.name = 'Semantic Key Attributes'
(9) post:
   let lastKey: Property = keyList->any() in
   let keyList1: Set(Property) = keyList->excluding(lastKey) in
  result.specification.body = OclPattern.Multiconcat
   (cl.name,
    '.allInstances()->forAll( i1, i2 | i1 <> i2 implies ',
    keyList1->iterate(key, acc: string '' |
      OclPattern.Multiconcat
        (acc, '(i1.', key.name, ' <> i2.', key.name, ') and ')),
    '(i1.', lastKey.name, ' <> i2.', lastKey.name, '))')
```

Fig. 6. Specification of pattern function *Create_Inv_SemanticKeyAttributes*

The specification for each pattern consists of three parts: pattern specific preconditions (1), general postconditions (2)-(5) and pattern specific postconditions (6)-(9).

The function specific preconditions describe which restrictions must be fulfilled when calling the pattern function. These preconditions must assure that the actual parameters conform to the specification pattern. The precondition (1) demands for instance that each element of *keyList* is an attribute of *cl*.

The general postconditions (2)-(5) are identical for all OCL pattern functions and represent in a way the main construction details. These postconditions (together with the functions signature) establish the following: The return of each pattern function is a UML model element of type *Constraint*. This constraint is added to the model (2) and is assigned to the model element which is the context of the constraint (3). The actual specification of the constraint is of type *OpaqueExpression* (4) and is edited in the language OCL (5).

The function specific postconditions (6)-(9) establish the following: (6) describes the constraint type (invariant, pre- or postcondition) of the returned constraint. (7) defines the context of the constraint (in our example the operation *cl*). The attribute *name* of *Constraint* is used in (8) to assign the pattern name to the constraint. The textual OCL representation of a constraint can be found in the attribute *body* of the constraint specification. Postconditions like (9) specify this textual representation by combining fixed substrings (e.g. `'.allInstances()'`) with the name of model elements which were supplied as pattern parameter values (e.g. `cl.name`). Note that we used in (9) a help function *OclPattern.Multiconcat* which concatenates a sequence of strings.

By defining OCL pattern functions for the specification patterns it became possible to formally specify the patterns completely and quite elegantly: the pattern parameters can be found as function parameters and the function specification (which uses again OCL) describes the prerequisites to apply the pattern and the properties of the constraint to be generated. One big advantage is that this approach only uses known specification techniques and does not require the invention of new ones. There is only one new class *OclPattern* that encapsulates the definition of all patterns. A more detailed description of the pattern specification, a comparison to other approaches and a discussion about the relationship to the UML metamodel can be found in [4].

7 Related Work

Due to its importance component specifications are discussed by many authors (e.g. [8,9,10,20,22] – for an overview compare e.g. [20]). Most current specification approaches identify the need for behavioral specifications and propose to use pre- and postconditions based on OCL [18]. Problems related with using OCL were so far only reported in the case studies [1,2] and the authors are not aware of any solution to this problem in the area of component specifications.

There are several publications outside the component area discussing the problems of editing OCL constraints manually [6,12,15]. There exist several approaches to simplify constraint writing: [12] develops an authoring tool that supports a developer with editing and synchronizing constraints in formal notation (OCL) and informal notation (natural language). [15] discusses an approach how to generate OCL expressions automatically. They constrain themselves, however, to the single use case of connecting two attributes within a UML model by an invariant. [11] discusses strategies to textually simplify OCL constraints that were generated by some algorithm. [21] develops an algorithm that allows in the analysis phase to transform use cases into class diagrams and OCL specifications. The author suggests that generation of OCL constraints might be possible but gives no details for it. [6] proposes a mechanism to connect design patterns with OCL constraint patterns which allows instantiating OCL constraints automatically whenever a design pattern is instantiated. This idea is very similar to ours but its realization can not be employed for specifying components (for details cf. [4]).

8 Summary

The paper discussed one of the current problems in component specifications: editing OCL constraints manually is time consuming and error-prone. As solution we proposed to utilize specification patterns for which OCL constraints can be generated automatically. For that we identified a collection of OCL specification patterns and presented a way to describe and formally specify these patterns. Such well-defined and formally specified patterns can be reused in component specification tools. Direction of future research include to gain further experience with the identified specification patterns (and extend the library if necessary) and with their usage in our component specification tool.

References

1. Ackermann, J.: Fallstudie zur Spezifikation von Fachkomponenten. In: Turowski, K. (ed.): 2. Workshop Modellierung und Spezifikation von Fachkomponenten. Bamberg (2001) 1-66 (In German)
2. Ackermann, J.: Zur Spezifikation der Parameter von Fachkomponenten. In: Turowski, K. (ed.): 5. Workshop Komponentenorientierte betriebliche Anwendungssysteme (WKBA 5). Augsburg (2003) 47-154 (In German)
3. 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 COEA 2005. Erfurt (2005) 41-56
4. Ackermann, J.: Formal Description of OCL Specification Patterns for Behavioral Specification of Software Components. In: Baar, T. (ed.): Proceedings of the MoDELS'05 Conference Workshop on Tool Support for OCL and Related Formalisms - Needs and Trends. Montego Bay, Jamaica (2005) 15-29
5. Ackermann, J., Turowski, K.: Specification of Customizable Business Components. In: Chroust, G.; Hofer, S. (eds.): Euromicro Conference 2003. Belek-Antalya, Turkey (2003) 391-394
6. Baar, T.; Hähnle, R.; Sattler, T.; Schmitt, P.H.: Entwurfgesteuerte Erzeugung von OCL-Constraints. In: Softwaretechnik-Trends 3 (2000) (In German)
7. Beugnard, A.; Jézéquel, J.-M.; Plouzeau, N.; Watkins, D.: Making Components Contract Aware. In: IEEE Computer 7 (1999) 38-44
8. Cheesman, J.; Daniels, J.: UML Components. Addison-Wesley, Boston (2001)
9. D'Souza, D.F.; Wills, A.C.: Objects, Components, and Frameworks with UML: The Catalysis Approach. Addison-Wesley, Reading (1998)
10. 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)
11. Giese, M.; Hähnle, R.; Larsson, D.: Rule-Based Simplification of OCL Constraints. In: Workshop on OCL and Model Driven Engineering at UML'2004. Lisbon (2004)
12. Hähnle, R.; Johannisson, K.; Ranta, A.: An Authoring Tool for Informal and Formal Requirements Specifications. In: Kutsche, R.-D.; Weber, H. (eds.): Fundamental Approaches to Software Engineering, 5th International Conference FASE. Grenoble (2002) 233-248
13. Hall, A.: Seven Myths of Formal Methods. In: IEEE Software 5 (1990) 11-19
14. 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) 235-243
15. Ledru, Y.; Dupuy-Chessa, S.; Fadil, H.: Towards Computer-aided Design of OCL Constraints. In: Grundspenkis, J.; Kirikova, M. (eds.): CAiSE Workshops 2004, Vol. 1. Riga (2004) 329-338
16. Meyer, B.: Applying "Design by Contract". In: IEEE Computer 10 (1992) 40-51

17. OMG (ed.): Unified Modeling Language: UML 2.0 Infrastructure Specification, 2004-10-16 URL: http://www.omg.org/technology/documents, Date of Call: 2005-09-09 (2004)
18. OMG (ed.): Unified Modeling Language: UML 2.0 OCL Specification, 2005-06-06. URL: http://www.omg.org/technology/documents, Date of Call: 2005-09-09 (2005)
19. OMG (ed.): Unified Modeling Language: UML 2.0 Superstructure Specification, 2005-07-04. URL: http://www.omg.org/technology/documents, Date of Call: 2005-09-09 (2005)
20. Overhage, S.: UnSCom: A Standardized Framework for the Specification of Software Components. In: Weske, M.; Liggesmeyer, P. (eds.): Object-Oriented and Internet-Based Technologies, Proceedings of the 5th Net'Object Days. Erfurt (2004)
21. Roussev, B.: Generating OCL specifications and class diagrams from use cases: A newtonian approach. In: Proceedings of 36th Annual Hawaii International Conference on System Sciences (HICSS'03). Big Island (2003)
22. Turowski, K. (ed.): Standardized Specification of Business Components: Memorandum of the working group 5.10.3 Component Oriented Business Application Systems. University of Augsburg (2002). URL: http://www.fachkomponenten.de. Date of Call: 2005-09-09
23. Wallnau, K.C.; Hissam, S.A.; Seacord, R.C.: Building Systems from Commercial Components. Addison-Wesley (2002)
24. Yellin, D.; Strom, R.: Protocol Specifications and Component Adaptors. In: ACM Transactions on Programming Languages and Systems 19 (1997) 292–333