

# Temporal OCL: Meeting Specification Demands for Business Components

Stefan Conrad<sup>+</sup> and Klaus Turowski<sup>\*</sup>

<sup>+</sup> *Ludwig-Maximilians-University Munich, Department of Computer Science, Oettingenstr. 67, D-80538 München, Germany, Tel.: +49 (89) 21 78-21 94, Fax: -21 92, E-Mail: conrad@dbs.informatik.uni-muenchen.de*

<sup>\*</sup> *Otto-von-Guericke-University Magdeburg, Institute for Technical and Business Information Systems, P.O. Box 41 20, D-39016 Magdeburg, Germany, Tel.: +49 (391) 67-1 83 86, Fax: -1 12 16, E-Mail: turowski@iti.cs.uni-magdeburg.de*

**Abstract.** Compositional plug-and-play-like reuse of black box components requires sophisticated techniques to specify components, especially if we combine third party components, which are traded on component markets, to customer-individual business application systems. As in established engineering disciplines like mechanical engineering or electrical engineering, we need a formal documentation of business components that becomes part of contractual agreements. Taking this problem as a starting point, we explain the general layered structure of software contracts for business components and show shortcomings of common specification approaches. Furthermore, we introduce a formal notation for the specification of business components that extends the Object Constraint Language (OCL) and that allows a broader use of the Unified Modeling Language (UML) with respect to the layered structure of software contracts for business components.

**Keywords:** Business Component; Formal Specification; Unified Modeling Language (UML); Object Constraint Language (OCL); Temporal Logic; Business Application System

## 1 Software Contracts for Business Components

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. By achieving this goal, advantages of individually programmed software with those of standardized off-the-shelf software could come together. In this context, we need compositional reuse techniques. Compositional reuse is a special reuse technique as generative techniques or code and design scavenging (Sametinger, 1997, pp. 25-28). The emphasis on compositional reuse stems from our *guiding model* which is the compositional plug-and-play-like reuse of black box components that are traded on a component market. In general, a guiding model is an ideal future state that might not completely be reached.

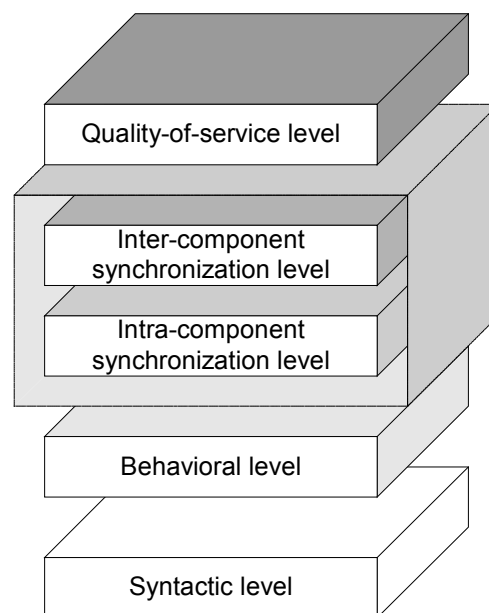
Corresponding to our guiding model, a company, which e.g. needs new software for stock keeping, could buy a suitable software component on the component market and further integrate it into its business application system with little effort. (Brown & Wallnau, 1996, pp. 11-14) explain the steps that are generally necessary to do so, e.g. technical and semantic adaptation or composition. Expected improvements, which should come along with using software components, concern cost efficiency, quality, productivity, market penetration, market share, performance, interoperability, reliability, or software complexity, cf. e.g. (Orfali, Harkey, & Edwards, 1996, S. 29-32).

According to (Fellner & Turowski, 2000, pp. 3-4), we understand by a *component* a reusable, self-contained and marketable software part, which provides services through a well-defined interface, and which can be deployed in configurations unknown at development time. A

*business component* is a component that implements a certain set of services out of a given business domain. Refer to (Szyperski, 1998, pp. 164-168) and (Fellner & Turowski, 2000) for an in deep discussion of various other component approaches given in literature.

To use business components according to our guiding model, it is necessary to standardize them, for a detailed discussion on standardizing business components cf. (Turowski, 2000). Furthermore, we have to describe their interface and behavior in a consistent and unequivocal way. In short, we have to *specify* them. Specification becomes more and more important with respect to third party composition of business components, since the specification might be the only available support for a composer who combines business components from different vendors to an application system.

*Software contracts* offer a good solution to meet the special requirements of specifying business components. Software contracts go back to MEYER, who introduced contracts as a concept in the *Eiffel* programming language. He called it *programming by contract* (Meyer, 1988) and extended it later to the concept of *design by contract* (Meyer, 1992). Furthermore, similar concepts are described in (Wilkerson & Wirfs-Brock, 1989) or (Johnson & Wirfs-Brock, 1990).



**Figure 1.** Software contract levels

Software contracts are obligations to which a service donator (e.g. a business component) and a service client agree. There, the service donator guarantees that

- a service it offers, e.g. *calculate balance* or *determine demand*,
- under certain conditions, which have to be met by the service client, e.g. the provision of data necessary to process the service,
- is performed in a guaranteed quality, e.g. with a predetermined storage demand or with an agreed response time, and
- that the service has certain external characteristics, e.g. the specified interface.

(Beugnard, Jézéquel, Plouzeau, & Watkins, 1999, pp. 38-40) describes a general model for software contracts for components with four tiers. The authors distinguish between syntactic, behavioral, synchronization, and quality-of-service level. Business components need to be specified on each of these levels.

Figure 1 shows contract levels according to (Turowski, 1999). By subdividing the synchronization level into inter- and intra-component synchronization level, as an extension to (Beugnard et al., 1999), this approach allows for an additional specification of a synchronization demand that exists between different business components.

At syntactic level basic agreements are concluded. Typical parts of these agreements concern names of services (offered by a business component), names of public accessible attributes, variables, or constant values, specialized data types (in common based upon standardized data types), signatures of services, as well as the declaration of error messages or exception signals. To do so, we use e.g. programming languages or *Interface Definition Languages* (IDL) like the IDL that was proposed by the Object Management Group (OMG) (OMG, 1998, S. 3.1-3.40). The resulting agreement guarantees that service client and service donator can communicate with each other. However, emphasis is put on enabling communication technically. Semantic aspects remain unconsidered.

Agreements at behavioral level serve as a closer description of a business component's behavior. They enhance the basic agreements of the syntactic level, which mainly describe the syntax of an interface. Agreements at syntactic level do not describe how a given business component acts in general or in borderline cases.

As an example, we could define an invariant condition for a business component *stock keeping* at behavioral level, which says that the reordering quantity for each (stock) account has to be higher than the minimum inventory level. Known approaches to specify behavior are based on approaches to *algebraic specification* of abstract data types, cf. e.g. (Ehrig & Mahr, 1985). To describe behavior, the specification of an abstract data type is extended by conditions. These conditions describe the abstract data type's behavior in general (as *invariant conditions*) or at specific times (*pre conditions* or *post conditions*). In general, conditions are formulated as equations, and as axioms they become part of the specification of an abstract data type (Ehrig & Mahr, 1985). The *Object Constraint Language* (OCL) (Rational Software et al., 1997a) is an example for a widespread notation to specify facts at the behavioral level. It complements the *Unified Modeling Language* (UML) (Rational Software et al., 1997b).

Agreements at intra-component synchronization level regulate the sequence in which services of a specific business component may be invoked, and synchronization demand between its services. Here, e.g., we may lay down that a minimum inventory level has to be set before it is allowed to book on a (stock) account for the first time, or that it is not allowed to carry through more than one bookkeeping entry at the same time for the same account.

At inter-component synchronization level we come to agreements that regulate the sequence in which services of *different* business components may be invoked. Here, e.g., we may define that a certain service, which belongs to a business component *shipping*, and which refers to a certain order, may only be processed after a service, which belongs to a business component *sales*, and which refers to the same order, has been processed at any time before. It is to note, that the differentiation between intra- or inter-component synchronization level depends on the identification of business components, but not on their granularity. The granularity of a business component depends on the number of services it offers.

There exist various approaches to specify business components at the synchronization levels. These approaches base, e.g., on using *process algebras*, *process calculi* (cf. e.g. (Hennessy, 1988)), or on using *temporal logics* (cf. e.g. (Alagar & Periyasamy, 1998, pp. 79-131)). In addition, (semi formal) graphical notations are in use. These are mostly graphical notations used in the context of business process modeling. Besides extended event-driven process chains (eEPC) (Keller, Nüttgens, & Scheer, 1992, pp. 32-35) and approaches that use eEPC as a basis, e.g. (Rittgen, 1999), Petri net based notations are in use, e.g. (Jaeschke, Oberweis, & Stucky, 1994).

As an extension to functional characteristics, we have to describe *non-functional* characteristics of business components. Non-functional characteristics are specified at the quality-of-service level. Examples for these characteristics are the distribution of the response time of a service or its availability. For further non-functional requirements and their definition cf. e.g. (Jalote, 1997, pp. 73-158).

## 2 Necessity of A Multi-Level Notation Standard

Using software contracts of the type explained in section 1 opens a way to a systematic specification of business components. Therefore, software contracts become a foundation for the third party composition of business components, which is conform to our guiding model. In an extreme case, employers of software components must be able to decide just with its specification about the way of its use.

Besides arranging the agreements' contents according to contract levels, in the context of systematic specification of business components it is helpful to use a well known and well-accepted formal notation, which can be used on more than one contract level. We call a notation *formal*, if syntax and semantics of the notation are unequivocal and consistent. For this reason, formal notations seem to be particularly suited to specify software contracts, which have to have these characteristics in order to be of use for third parties.

The OMG IDL, as part of the *Common Object Request Broker Architecture* (CORBA) (OMG, 1998), gains more and more acceptance as a standardized notation for the syntactic level. It uses a so-called *IDL compiler* to translate the interface's specification into concrete programming languages. UML together with the OCL is the addition (recommended by the OMG) to specify facts that belong to the behavioral level. Furthermore, the UML (together with the OCL) is especially recommended to specify components, e.g. (Allen & Frost, 1998) or (D'Souza & Wills, 1999). However, the OCL is only conditionally suited to specify facts at the synchronization level(s) as well. Taking this problem as a starting point, we propose a way to extend the OCL with some additional temporal operators to be able to formally specify facts at the synchronization levels as well.

However, we would like to point out that some authors tend to criticize upon the formal specification of (parts of) business application systems. Their main arguments are comparably higher effort and decreased general understandability. As an example for the weaknesses of formal approaches, they often discuss the algebraic specification of abstract data types, cf. e.g. (Biethahn, Mucksch, & Ruf, 1991, pp. 288-291) and the references given there. It remains to note that these authors also mention the very good separation of inside view and outside view, as important advantages of the algebraic specification of abstract data types. This is a fact that is from growing importance with respect to the specification of black box components.

*Operational* and *verbal* specification are discussed as more practicable alternatives to algebraic specification. For operational specification, specification is done using declarative capabilities of programming languages (Ferstl & Sinz, 1998, pp. 293-294). This way, syntactic and behavioral level may be specified – dependent on the chosen programming language. For verbal specification, natural language is used. Due to its inherent fuzziness, natural languages are only conditionally suited to specify business components. For example, they may be used in addition to a formal specification or together with specialized methods like norm language reconstruction (Ortner, 1997).

## 3 Using OCL to Specify Business Components

In the following, we explain an example to show how to use the OCL to specify business components on the behavioral level. In order to complete the example with respect to the contract levels given in section 1, we first explain the agreements necessary of the syntactic

level. For the example, we use the OMG IDL as interface definition language (OMG, 1998, pp. 3.1-3.40).

```

interface OrderProcessing {
    ...
    struct OrderPosition {
        double Quantity;
        double PiecePrice;
        double Discount;
    };
    ...
    struct Order {
        ...
        boolean                TechnicallyPracticable;
        boolean                Delivered;
        double                 InvoiceAmount;
        double                 Discount;
        ...
        sequence <OrderPosition> OrderPositions;
    };
    ...
    void AcceptCustomerOrder(in Order a);
    void CancelOrder(in Order a);
    void PrintInvoice(in Order a);
    ...
};

interface ProductionPlanning {
    ...
    ProductionPlan RoughPlanning(in Period p);
    ...
};

interface ProductionControl {
    ...
    ProductionsPlan Scheduling(in Period p, in ProductionPlan pp);
    ...
};

```

**Figure 2:** Specification of business components at the syntactic level

Figure 2 shows examples for the specification of the interface of different business components at the syntactic level. The figure depicts parts of the interfaces of the business components *OrderProcessing*, *ProductionPlanning*, and *ProductionControl*. The business components support business tasks from the area of production planning and control (PPC), cf. e.g. (Scheer, 1994).

First, the name of the service donator is defined with the keyword `interface`. This keyword creates a name space, which allows for an unequivocal definition of contained names. `OrderProcessing::PrintInvoice` e.g., indicates that a service *PrintInvoice* should be invoked that is part of a business component *OrderProcessing*.

In addition, we need to define data types, structured data types, and exceptions, which we will not explain further in the context of our example.

In figure 3 we extend our example to specify requirements at behavioral level by using the OCL (Rational Software et al., 1997a). The OCL is part of the UML. Furthermore, it was adopted by the OMG as standardized notation. With this, the OCL is the recommended extension of the OMG IDL to specify requirements at behavioral level.

First of all, we fix the context to which the respective specification refers. We mark the context by underlining it. The first condition in figure 3, e.g., refers to the business component *OrderProcessing* as a whole. Conditions appear either as pre conditions (keyword `pre`), as post condition (keyword `post`), or as invariant condition (no keyword).

For the purposes of our example, order processing encompasses the management of orders. The symbolic term *Order* references orders. Thus, the first invariant condition ensures that all

orders, which are hold by the business component *OrderProcessing*, are technically practicable.

```

OrderProcessing
self.Order->forall(a:Order | a.TechnicallyPracticable = True)

OrderProcessing::PrintInvoice(at:Order)
pre : self.Order->exists(a:Order | a = at and at.Delivered = True)
post: at.InvoiceAmount = at.OrderPositions->iterate(p:Position; b:Amount = 0 |
    b + p.Quantity * p.PiecePrice * (1 - p.Discount)
) * (1 - at.Discount)

```

**Figure 3.** Examples for the specification of business components at behavioral level using OCL

The other requirements relate to the service *PrintInvoice*. For this reason, `::` restricts the conditions' context. In addition, parameters may be enumerated to describe a service's behavior in more detail. Take, e.g., the service *PrintInvoice*. In order to specify it in more detail, we use the typed parameter *at*, which is of type *order*. (The types that we use in the example correspond to those defined in figure 2.)

We use a pre condition for the service *PrintInvoice*. It ensures that printing an invoice is allowed, if and only if the corresponding order was delivered before. Furthermore, there is a post condition that explains in detail, how the invoice amount was calculated.

On principle, all requirements of that kind are local to their respective contexts. The context may be the business component that offers the respective service. Therefore, we could suppose, that, e.g., pre conditions for services, which are part of one single business component, may relate to services or objects of the same business component. However, business components are not isolated, but they collaborate with each other. For that reason, services and properties of other business components are published as interfaces. Thus, all published services and properties of other business components become part of the context of one particular business component. Furthermore, it is possible to refer to other business components while specifying pre, post, and invariant conditions for a particular business component. Thus, characteristics of one particular business component may influence (or restrict) the behavior of other business components.

The possibility to describe characteristics that spread to different business components raises the question in which business component these kinds of characteristics have to be specified and whether it is necessary to specify these characteristics redundantly. However, these methodic aspects go beyond the concern of our contribution. For this reason we omit a detailed discussion of the mentioned aspects.

#### 4 Temporal Extension of OCL

In the previous section, we discussed in which way OCL could be employed for specifying business components. OCL seems to be an ideal approach to describing properties of business components declaratively and independent of specific implementations. Thereby, OCL can be used as an integral part of software contracts. OCL allows describing properties of states (which must hold for each single state of the system or component) and to describe pre and post conditions of services offered by a business component. By means of pre and post conditions we can restrict the applicability (or executability) of services. Furthermore, the result of a service (the effect of its execution) can be specified by referring to the state of its invocation (using `@pre`).

Thus, introducing `@pre` and `@post` for explicitly referencing values of the states directly before and after a service execution allows to specify a certain kind of conditions. In the context of database systems such conditions are usually called transitional integrity constraints (cf. e.g. (Lipeck, Gertz, & Saake, 1994)). Following this comparison with integrity constraints in

database systems, OCL, of course, also allows to specify static integrity constraints (without @pre and @post). In this widely used classification of integrity constraints the class of temporal constraints remains which cannot be described by means of OCL (except of a few cases where a translation into transitional constraints is possible). Temporal constraints do not only describe state transitions triggered by calling services but also complete lifespans of objects or large parts of evolution within a business component.

Some restricted kinds of temporal constraints can also be described by means of state charts (being a part of UML for modeling behavior of objects) or other models of state machines. However, certain temporal constraints cannot be represented by state machines at all, for instance the constraint that after executing a certain service A another service B cannot be invoked unless a service C has been executed. Of course, we may find a state machine fulfilling this constraint, however, it is always a concrete implementation restricting the behavior of the system more than the temporal constraint requires. For our purposes, we do not want to specify certain implementation but the general properties business components have to meet.

Temporal constraints are a means to declaratively describe properties of components at the interface level. The required view from outside onto components is essential why state machines are not the adequate level of description for us. In addition, their operational character is not appropriate for that level of specification. In consequence, we here focus on temporal integrity constraints as means of description.

Within the application area introduced before we may have the following temporal integrity constraints as examples:

- A service *Scheduling* can only provide a result for a certain period of time, if a service *RoughPlanning* was already executed before for the same period.
- The execution of a service *PrintInvoice* for a certain order requires that exactly this order has been entered using a service *AcceptCustomerOrder* and that in-between this order has not been canceled by executing the service *CancelOrder*.

In addition we could require that after executing the service *AcceptCustomerOrder* an invoice must be written for that order by means of the service *PrintInvoice* or that the order has eventually to be canceled by invoking the service *CancelOrder*.

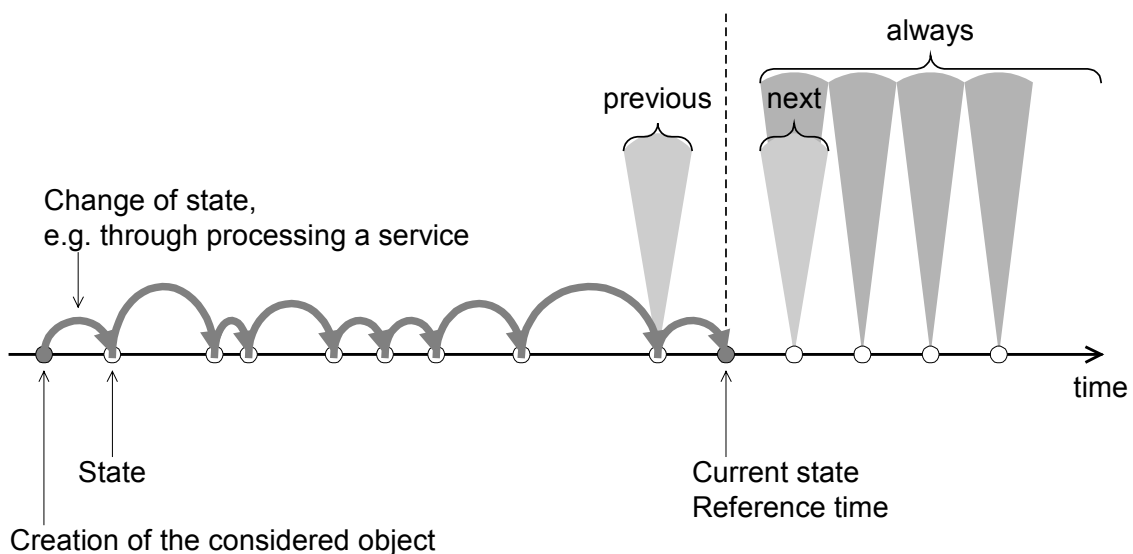


Figure 4. State based specification using (linear) temporal logics

For specifying such properties, it seems to be a good approach to extend OCL. This is motivated by the fact that OCL is a standardized notation based on a well-known declarative for-

malism having a clear (and formal) semantics. We do not intend to create an own specification formalism for temporal integrity constraints. Therefore, it is reasonable to look for a minimal but sufficiently expressive extension of OCL being consistent with OCL.

The basic possibilities to describe temporal properties are depicted in figure 4. On the time axis the different states of an object (which is subject of our description) at different instants of time are given. Starting from the current state of this object, different statements can be made about this object. For instance, we can describe properties of the state directly before or after the current one (by means of the temporal operator *previous* and *next*, resp.). Using the (future tense temporal) operator *always* we can state properties of all future states. The current state has been reached by executing all state changing operations (services in our context) in their temporal order starting with the initial state of the object (the initial state is usually given when creating the object). The temporal distances between all pairs of consecutive states need not to be the same, in this way, we only use an abstract notion of discrete time representing the order of changes.

A large number of approaches to describing such temporal properties is based on temporal logics (for a survey see e.g. (Manna & Pnueli, 1992) or (Emerson, 1990)). In (Lipeck & Saake, 1987) a temporal logic for formulating temporal integrity constraints was developed which then has been adapted and extended for the object-oriented modeling language TROLL (Jungclaus, Saake, Hartmann, & Sernadas, 1996). The temporal operators by which we extend OCL and for which all semantic foundations can also be found in (Lipeck & Saake, 1987) are as follows:

Past tense temporal operators:

- `sometime_past  $\varphi$`   
Starting from the current state (i.e. the state which is currently observed) sometime in the past  $\varphi$  must have been valid (i.e. there is a past state in which  $\varphi$  held).
- `always_past  $\varphi$`   
Starting from the current state  $\varphi$  was valid always in the past (i.e. in all past states).
- `$\varphi$  always_since_last  $\psi$`   
Starting from the last state in which  $\psi$  held for the last time,  $\varphi$  held in all states up to the current one.
- `$\varphi$  sometime_since_last  $\psi$`   
Since the state in which  $\psi$  held for the last time there was a state (before the current state) in which  $\varphi$  held.

Future tense temporal operators:

- `sometime  $\varphi$`   
Starting from the current state there will be (at least) one future state in which  $\varphi$  will be valid.
- `always  $\varphi$`   
Starting from the current state in all future states  $\varphi$  will hold.
- `$\varphi$  until  $\psi$`   
Starting from the current state in all future state  $\varphi$  will hold until there is a state in which  $\psi$  is fulfilled.
- `$\varphi$  before  $\psi$`

Starting from the current state there will be a future state in which  $\varphi$  will be valid before a state will be reached in which  $\psi$  will hold.

Beside these past tense and future tense temporal operators which are always interpreted relative to a current state we need a special operator for referring to the initial state (of the system or business component):

- `initially  $\varphi$`

In the initial state  $\varphi$  holds.

By means of this operator, it is possible to specify the initial state of the system (i.e. to provide initial values for some state variables).

Before we consider examples for temporal constraints in our application area using the temporal extension of OCL, the necessity of offering past tense and future tense temporal operators has to be discussed. Taking a puristic view one could claim that one kind of temporal operators (i.e. past tense or future tense) would be sufficient. Although this is already not completely right with regard to expressive power, the main reason for having both kinds of operators is a methodical one. Using each kind of temporal operators in an adequate way essentially improves the readability and, thereby, the comprehensibility of specifications. For instance, pre conditions for services (i.e. constraints restricting the applicability of services) should be formulated only by using past tense temporal operators. It is obvious that the execution of a service must not depend on future states.

By allowing post conditions to include future tense temporal operators we slightly change or extend the notion of post condition. In the literature, and in particular for object oriented languages, the notion of post condition refers usually to a property fulfilled by the state yielded by executing a method. A temporal logic formula as post condition also refers to other future states. Nevertheless, from a logical point of view this property formulated in temporal logic can also be considered as a property of that state.

A further important issue is that the grammar of OCL has to be extended corresponding to the temporal operators we add. Considering the grammar for OCL given in (Rational Software et al., 1997a, pp. 31-32) we only need very few minor changes and additions, in detail these are as follows:

- An additional alternative `temporalExpression` is introduced into the rule for `relationalExpression`.
- For `temporalExpression` a rule is added in which temporal expressions are constructed in two different ways - using either a unary temporal operator like (`always`) or a binary one (like `until`).
- In two additional rules the unary and binary temporal operators are defined (`UnaryTemporalOperator` and `BinaryTemporalOperator`).

The complete extended grammar is given as appendix.

Obviously, our extension seamlessly fits into the existing grammar for OCL without requiring significant changes. Thereby, integration into already existing tools supporting OCL should not cause severe problems.

Figure 5 shows formulations of the temporal (integrity) constraints introduced verbally at the beginning of this section. The temporal extension of OCL sketched before is now used to express these constraints. The first statement is a pre condition for executing the service *Scheduling* for a certain period of time. It is required that for the same period of time a service *RoughPlanning* was already executed sometime before. Here, we assume that the service *RoughPlanning* provided by another business component is known in the component *ProductionPlanning* by declaring its interface description.

```

ProductionPlanning::Scheduling(p:Period)
pre : sometime_past( RoughPlanning(p) )

OrderProcessing::PrintInvoice(at:Order)
pre : sometime_past( AcceptCustomerOrder(at) ) and
not( CancelOrder(at) sometime_since_last AcceptCustomerOrder(at) )

OrderProcessing::AcceptCustomerOrder(at:Order)
post: sometime( PrintInvoice(at) ) or sometime( CancelOrder(at) )

```

**Figure. 5:** Specification of temporal properties using the extended OCL

The second statement in Figure 5 is a pre condition for the service *PrintInvoice* in the business component *OrderProcessing*. This pre condition expresses that before this service can be invoked the service *AcceptCustomerOrder* must have been executed for the same order and that since that acceptance of this order no cancellation of this order has been occurred (by executing the service *CancelOrder*).

The third temporal property is a post condition for the service *AcceptCustomerOrder*. In addition to the previous statement it is required that after accepting an order by a customer sometime later an invoice has to be printed for exactly this order by executing the service *PrintInvoice* or that this order must eventually be cancelled by executing the service *CancelOrder*.

The second and third statement express different properties. On the one hand, the pre condition for *PrintInvoice* does not forbid that the service *AcceptCustomerOrder* can be executed without that an invoice will ever be printed or a cancellation will ever occur for that order. On the other hand, the post condition for *AcceptCustomerOrder* does not exclude the execution of the service *PrintInvoice* for a certain order although this order has never accepted by means of the service *AcceptCustomerOrder*.

Finally, we have to discuss the issue of formal semantics for this extension of OCL by temporal operators. As already mentioned before the semantic foundations (i.e. a complete definition of the formal semantics for a temporal logic with past tense and future tense temporal operators) are given for instance in (Lipeck & Saake, 1987) such that we refrain from repeating these definitions here. In contrast to (Lipeck & Saake, 1987), we did not introduce the operators *next* (referring to the subsequent state) and *previous* (referring to the previous state). This is due to the well-known semantic problems, which are caused by these two operators in case of composing independently specified systems of components. In general, we implicitly obtain concurrent processes in the composed system where no global synchronization of local states is given. As a consequence, considering a common global state *next* operators in the specifications of different components may for instance refer to different local states which do not necessarily belong to one global state (cf. also (Conrad, 1995) or (Mokkedem & Méry, 1994)). Although there are several proposals for solving this kind of problems (beside the references mentioned before see also (Conrad, 1996), (Sørensen, Hansen, & Løvengreen, 1994)), they all are not yet developed so far such that a full semantic compositionality of specifications of components is given without causing methodical restrictions in specifying single components.

## 5 Conclusions and Outlook

The usage of business components offers a possibility to customize business application systems incorporating the advantages of standardized software and individually developed software. However, this requires the standardization of business components and, in consequence, a specification of these components. After having derived basic requirements for specifications of business components from investigating the paradigm of software contracts, we presented a proposal for extending OCL by temporal operators. Based on a widespread notational

standard it is now possible to specify across several software contract levels avoiding a change of methods and respecting the particular requirements of business components.

It should be noted that our proposal - in contrast to OCL itself - is not standardized and, therefore, it has the same status as other proprietary notations or extension of notations. Considering the fact that such extensions by temporal properties are indispensable for specifying business components, our proposal might be an essential first step towards a later standardization.

We obtain first practical experiences towards practicality and usability of our approach in the context of in house projects, which concerned the development of business components for the application domain production planning and control. After an introductory tutorial, the notation's extension was well accepted by the project's participants. Problems that arose had their reason basically in a wrong understanding of dependencies of the application domain. Consequently, in some cases we could observe a not adequate specification.

## References

- Alagar, V. S., & Periyasamy, K. (1998). *Specification of Software Systems*. New York: Springer.
- Allen, P., & Frost, S. (1998). *Component-Based Development for Enterprise Systems: Applying The Select Perspective*. Cambridge: Cambridge University Press.
- Beugnard, A., Jézéquel, J.-M., Plouzeau, N., & Watkins, D. (1999). Making Components Contract Aware. *IEEE Computer*, 32(7), 38-44.
- Biethahn, J., Mucksch, H., & Ruf, W. (1991). *Ganzheitliches Informationsmanagement: Daten- und Entwicklungsmanagement*. (Vol. 2). München: Oldenbourg. (*Holistic Information Management: Managing Data and Development*)
- Brown, A. W., & Wallnau, K. C. (1996). Engineering of Component-Based Systems. In A. W. Brown (Ed.), *Component-Based Software Engineering: Selected Papers from the Software Engineering Institute* (pp. 7-15). Los Alamitos, California: IEEE Computer Society Press.
- Conrad, S. (1995). *Compositional Object Specification and Verification*. Paper presented at the International Conference on Software Quality (ICSQ'95), Maribor.
- Conrad, S. (1996). A Basic Calculus for Verifying Properties of Interacting Objects. *Data and Knowledge Engineering*, 18(2), 119-146.
- D'Souza, D. F., & Wills, A. C. (1999). *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Reading: Addison-Wesley.
- Ehrig, H., & Mahr, B. (1985). *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Berlin: Springer.
- Emerson, E. A. (1990). Temporal and Modal Logic. In J. v. Leeuwen (Ed.), *Handbook of Theoretical Computer Science* (Vol. B, pp. 995-1072). Amsterdam: Elsevier Science Publishers, North-Holland.
- Fellner, K., & Turowski, K. (2000). *Classification Framework for Business Components*. Paper presented at the 33rd Annual Hawaii International Conference On System Sciences, Maui, Hawaii.
- Ferstl, O. K., & Sinz, E. J. (1998). *Grundlagen der Wirtschaftsinformatik*. (3 ed.). (Vol. 1). München: Oldenbourg. (*Foundations of Business Information Systems*)
- Hennessy, M. (1988). *Algebraic Theory of Processes*. Cambridge: MIT Press.
- Jaeschke, P., Oberweis, A., & Stucky, W. (1994). *Deriving Complex Structured Objects for Business Process Modelling*. Paper presented at the 13th Int. Conf. on the Entity-Relationship Approach, Manchester.
- Jalote, P. (1997). *An Integrated Approach to Software Engineering*. New York: Springer.
- Johnson, R. E., & Wirfs-Brock, R. J. (1990). Surveying Current Research in Object-Oriented Design. *Communications of the ACM*, 33(9), 104-124.
- Jungclaus, R., Saake, G., Hartmann, T., & Sernadas, C. (1996). Troll: A Language for Object-Oriented Specification of Information Systems. *ACM Transactions on Information Systems*, 14(2), 175-211.
- Keller, G., Nüttgens, M., & Scheer, A.-W. (1992). Planungsinseln - Vom Konzept zum integrierten Informationsmodell. *HMD*, 29(168), 25-39. (*Planning Islands - From Concept to Integrated Information Model*)
- Lipeck, U. W., Gertz, M., & Saake, G. (1994). Transitional Monitoring of Dynamic Integrity Constraints. *Bulletin of the IEEE Technical Committee on Data Engineering*, 17(2), 38-42.

- Lipeck, U. W., & Saake, G. (1987). Monitoring Dynamic Integrity Constraints Based on Temporal Logic. *Information Systems*, 12(3), 255-269.
- Manna, Z., & Pnueli, A. (1992). *The Temporal Logic of Reactive and Concurrent Systems: Specification*. (Vol. 1). Berlin: Springer.
- Meyer, B. (1988). *Object-Oriented Software Construction*. Englewood Cliffs: Prentice Hall.
- Meyer, B. (1992). Applying "Design by Contract". *IEEE Computer*, 25(10), 40-51.
- Mokkedem, A., & Méry, D. (1994). *A Stuttering Closed Temporal Logic for Modular Reasoning about Concurrent Programs*. Paper presented at the First International Conference on Temporal Logic (ICTL'94), Bonn.
- OMG (Ed.). (1998). *The Common Object Request Broker: Architecture and Specification (Revision 2.2)*: OMG.
- Orfali, R., Harkey, D., & Edwards, J. (1996). *The Essential Distributed Objects Survival Guide*. New York: John Wiley & Sons.
- Ortner, E. (1997). *Methodenneutraler Fachentwurf*. Stuttgart: Teubner. (*Method Invariant Design*)
- Rational Software, Microsoft, Hewlett-Packard, Oracle, Sterling Software, MCI Systemhouse, Unisys, ICON Computing, IntelliCorp, i-Logix, IBM, ObjecTime, Platinum Technology, Ptech, Taskon, Reich Technologies, & Softeam. (1997a). *Object Constraint Language Specification: Version 1.1, 1 September 1997*. Available: <http://www.rational.com/uml> [1999, 04-17].
- Rational Software, Microsoft, Hewlett-Packard, Oracle, Sterling Software, MCI Systemhouse, Unisys, ICON Computing, IntelliCorp, i-Logix, IBM, ObjecTime, Platinum Technology, Ptech, Taskon, Reich Technologies, & Softeam. (1997b). *UML Notation Guide: Version 1.1, 1 September 1997*. Available: <http://www.rational.com/uml> [1999, 04-17].
- Rittgen, P. (1999). *Objektorientierte Analyse mit EMK*. Paper presented at the Modellierung betrieblicher Informationssysteme: MobIS-Fachtagung 1999 (MobIS'99), Bamberg. (*Object-Oriented Analyses Using EMK*)
- Sametinger, J. (1997). *Software Engineering with Reusable Components*. Berlin: Springer.
- Scheer, A.-W. (1994). *Business Process Engineering: Reference Models for Industrial Enterprises*. (2 ed.). Berlin: Springer.
- Sørensen, M. U., Hansen, O. E., & Løvengreen, H. H. (1994). *Combining Temporal Specification Techniques*. Paper presented at the First International Conference on Temporal Logic (ICTL'94), Bonn.
- Szyperski, C. (1998). *Component Software: Beyond Object-Oriented Programming*. (2 ed.). Harlow: Addison-Wesley.
- Turowski, K. (1999). *Standardisierung von Fachkomponenten: Spezifikation und Objekte der Standardisierung*. Paper presented at the 3. Meistersingertreffen, Schloss Thurnau. (*Standardizing Business Components: Specification and Objects of Standardization*)
- Turowski, K. (2000). Establishing Standards for Business Components. In K. Jakobs (Ed.), *Information Technology Standards and Standardisation: A Global Perspective* (pp. 131-151). Hershey: Idea Group Publishing.
- Wilkerson, B., & Wirfs-Brock, R. J. (1989). A Responsibility-Driven Approach. *SIGPLAN Notices*, 24(10), 72-76.

## Appendix (Grammar of Temporal OCL)

```
expression                := logicalExpression
ifExpression              := "if" expression
                           "then" expression
                           "else" expression
                           "endif"

logicalExpression        := relationalExpression
                           ( logicalOperator relationalExpression )*

relationalExpression     := temporalExpression
                           | additiveExpression
                           ( relationalOperator additiveExpression )?

temporalExpression       := unaryTemporalOperator logicalExpression
                           | logicalExpression
                           binaryTemporalOperator logicalExpression

additiveExpression       := multiplicativeExpression
                           ( addOperator multiplicativeExpression )*

multiplicativeExpression := unaryExpression
                           ( multiplyOperator unaryExpression )*

unaryExpression          := ( unaryOperator postfixExpression )
                           | postfixExpression

postfixExpression        := primaryExpression ( "." | "->" ) featureCall )*
primaryExpression        := literalCollection
                           | literal
                           | pathName timeExpression? qualifier?
                           featureCallParameters?
                           | "(" expression ")"
                           | ifExpression

featureCallParameters    := "(" ( declarator )? ( actualParameterList )? ")"
literal                  := <STRING> | <number> | "#" <name>
enumerationType          := "enum" "{" "#" <name> ( "," "#" <name> )* "}"
simpleTypeSpecifier       := pathTypeName
                           | enumerationType

literalCollection        := collectionKind "{" expressionListOrRange? "}"
expressionListOrRange    := expression
                           ( ( "," expression )+
                           | ( ".." expression )
                           )?

featureCall               := pathName timeExpression? qualifiers?
                           featureCallParameters?

qualifiers                := "[" actualParameterList "]"
declarator                := <name> ( "," <name> )*
                           ( ":" simpleTypeSpecifier )? "|"

pathTypeName              := <typeName> ( "::" <typeName> )*
pathName                  := ( <typeName> | <name> )
                           ( "::" ( <typeName> | <name> ) )*

timeExpression           := "@" <name>
actualParameterList      := expression ( "," expression )*
logicalOperator           := "and" | "or" | "xor" | "implies"
collectionKind           := "Set" | "Bag" | "Sequence" | "Collection"
relationalOperator       := "=" | ">" | "<" | ">=" | "<=" | "<>"
addOperator               := "+" | "-"
multiplyOperator          := "*" | "/"
unaryOperator             := "-" | "not"
```

```

typeName      := "A"- "Z" ( "a"- "z" | "0"- "9" | "A"- "Z" | "_" ) *
name          := "a"- "z" ( "a"- "z" | "0"- "9" | "A"- "Z" | "_" ) *
number        := "0"- "9" ( "0"- "9" ) *
string        := "" ( ( ~ [ " " , "\\", "\n", "\r" ] )
                    | ( "\\"
                        ( [ "n", "t", "b", "r", "f", "\\", " ", "\"" ]
                          | [ "0"- "7" ] ( [ "0"- "7" ] ) ?
                          | [ "0"- "3" ] [ "0"- "7" ] [ "0"- "7" ]
                        )
                      )
                    ) *
""
unaryTemporalOperator := "sometime_past" | "always_past"
                       | "sometime" | "always" | "initially"
binaryTemporalOperator := "sometime_since_last" | "always_since_last"
                       | "until" | "before"

```