

On Message Specifications in OCL¹

Marcel Kyas²

*Institute for Computer Science and Applied Mathematics
Christian-Albrechts-Universität zu Kiel
Germany*

Frank S. de Boer³

*CWI Amsterdam
The Netherlands*

Abstract

The object constraint language (OCL) is the established language for specifying of properties of objects and object structures. Recently an extension of OCL has been proposed for the specification of messages sent between objects.

In this paper we present a generalization of this extension which allows additionally to specify causality constraints. From a pragmatic point of view, such causality constraints are needed to express, for example, that each acknowledgment must be preceded by a matching request, which is frequently required by communication protocols.

Our generalization is based on the introduction of histories into OCL. Histories describe the external behavior of objects and groups of objects. Moreover, to reason compositionally about the behavior of a complex system we distinguish between *local* specifications of a single object and *global* specifications describing the interaction between objects. These two types of specifications are expressed in syntactically different dialects of OCL. Our notion of compositionality, which is formalized in this paper by a compatibility predicate on histories, allows the verification of models during the early stages of a design.

Key words: OCL, Compositional Verification

¹ Part of this work has been financially supported by IST project Omega (IST-2001-33522) and NWO/DFG project Mobi-J (RO 1122/9-1, RO 1122/9-2)

² <mailto:mky@informatik.uni-kiel.de>

³ <mailto:frb@cwi.nl>

1 Introduction

Today, UML [15,16] and its textual specification language OCL [19,3] are widely used as specification and modeling languages for object-oriented systems. OCL is used to constraining object structures in UML. Constraints on an object structure are invariants over a state of a system. Such invariants use an object's attributes and the relationships between objects.

The behavioral concepts in OCL are pre- and postconditions of operations. The additional behavioral concept of a *message expression* has been introduced in OCL 2.0 [3]. Message expressions are expressions on messages sent by a particular object. As we shall demonstrate this imposes a real restriction.

We introduce a general framework for the behavioral specification of objects, a framework for modular specifications, and a compositional verification method for OCL. Behavior is described in terms of messages and histories of events.

OCL 2.0 introduces a history as a sequence of local snapshots as part of the interpretation of an object's valuation [3, pp. 5-4–5-5]. This history is not the kind of history we describe here. The history defined in OCL 2.0 is only part of the semantic domain of OCL and has no *syntactic* representation in OCL itself. It is therefore impossible to use this history in an OCL specification. In OCL 2.0 there is no type which is interpreted by `Sequence(LocalSnapshot)`, and there is no expression whose value is the history of an Object.

Adhering to the encapsulation principle of object-oriented programming we separate specifications into a *local* part describing the behavior of an object, and a *global* part describing the collaboration between different objects to achieve a common goal. More precisely, a local specification consists of a specification of the internal structure of an object and a specification of the observable behavior of an object, its *local* history. A global specification specifies how the objects are associated to each other and how they *exchange* messages using a *global* history.

The compositional verification method is based on a compatibility predicate over local histories and the global history, which states that the composition of objects is feasible and the globally specified behavior is achieved. The compatibility predicate introduced in this paper is a generalization of the compatibility predicate for CSP described in [22] to object oriented systems. The verification step of checking the compatibility predicate relies only on the objects' observable behavior and not on any specification of their internal structure. This enables the use of the compatibility test to identify design errors during early stages of the design.

The specification language used to specify a program's components may only use predicates over their *observable behavior*. According to the encapsulation principle, a *behavioral* specification language should never state properties of the interior construction of the constituent components, for example

the underlying execution platform.

The encapsulation mechanisms of object-oriented design supports the decomposition in a natural way. Object-oriented design makes the interface of the parts of a large system explicit, and aggregation is a way to group objects into larger parts. However, OCL does not enforce this encapsulation. To allow this we require the separation of specifications into local and global properties [1,2]:

- (i) A *local* specification is an OCL constraint on the local attributes and the local history of events of a single object, only.
- (ii) A *local behavioral* specification is a local specification which only constrains the local history of an object and does not refer to the object's internal structure.
- (iii) A *global* specification is an OCL constraint on the links, i.e., references, *between* objects.

Local behavioral specifications and global specifications are used to separate concerns: The local behavioral specification is used to specify the behavior of a program's constituents whereas the global specification is used to specify how these constituents are put together. The local specification constrains the interior construction of an object. For a compositional specification we only need local behavioral specifications and global specifications.

The remainder of this paper is structured as follows: In the next section we summarize the current proposal on extending OCL with message expressions and demonstrate a weakness of their approach when applied to message expressions. In Section 3 we describe the notion of events and histories used in our extension of OCL. We identify the *observable behavior* of an object for the assertional specification of objects. We use the traditional choice of messages sent and received by an object. In Section 4 we describe how our method facilitates compositional specifications and reasoning. We explain the distinction between local specifications, local behavioral specifications, and global specifications. In Section 5 we describe our compositional verification method. Finally, we draw some conclusions and compare our results to related work. We use the terminology defined in [3,15,16].

2 State of the Art and Motivation

We describe the means of specifying interactions between objects in OCL 2.0, and in what respect they are not sufficient from a pragmatic point of view. To explain this, we introduce the *Sieve of Erasthostenes* as an example, and follow the ideas presented in [1].

The model consists of the two classes *Generator* and *Sieve* (see Figure 1). Exactly one instance, the root object, of the class *Generator* is present in the model. The generator creates an instance of the *Sieve* class. Then it sends

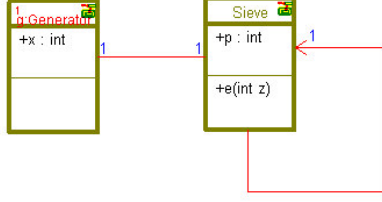


Fig. 1. Class Diagram of the Sieve

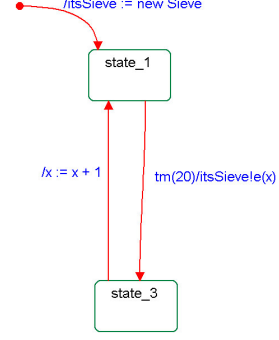


Fig. 2. Generator's State Machine

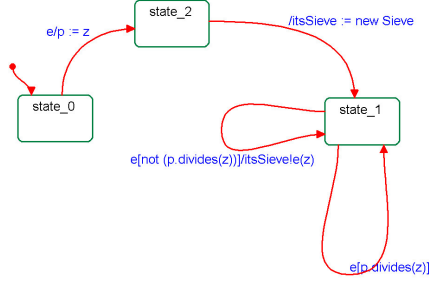


Fig. 3. Sieve's State Machine

```

context Generator
inv:  $x > 1$ 
context Sieve
inv:  $\text{Integer}\{2..(p-1)\} \rightarrow \text{forAll } (i \mid p \bmod i < 0)$ 
    
```

Fig. 4. Specification of the Sieve

the new instance natural numbers in increasing order, see Figure 2.⁴ The association from Generator to Sieve is called *itsSieve*.⁵

Each instance of Sieve has an attribute p , which holds a number. It receives sequence of integers i and if i is not divisible by p , then it sends i to its successor. Otherwise it discards the number and awaits the next number. The behavior is shown in Fig. 3.

We want to specify this behavior in OCL and try to prove that the Sieve described by the state machines in Fig. 2 and Fig. 3 are indeed implementations of the specified behavior. With this specification we want to abstract from the state machines, which already provide an implementation of the sieves behavior. The property we want to prove is: The value of each attribute p of an instance of Sieve is a prime number. The OCL representation is shown in Figure 4.

Whether this property is satisfied by the model depends on the behavior of the other objects within the system. For example, we require that the generator does not send the value 1 to a sieve object, that the generator sends numbers in monotonically increasing order, and that the messages sent to a sieve object are *received* in monotonically order. This last requirement is the

⁴ A transition is labeled by a transition is labeled with an event e , which triggers the transition, a guard g on the object's state, which has to be satisfied, if the transition is to be taken, and an action a written in a ALGOL like language. We write this label as $e[g]/a$, or shorter e/a , if the guard g is *true*.

⁵ If no role name for an association is mentioned, a default name is used by concatenating *its* and the associated class name.

reason for our extension.

OCL 2.0 introduces the two operators $\hat{}$ and $\hat{\hat{}}$ for reasoning about messages. The first one, $\text{o}\hat{\text{msg}}(\text{e})$, reads: “The contextual instance has sent a message `msg` to an object `o` with parameter values `e`.” It is a predicate stating that a message has been sent during the execution of an operation. As such it should only be used in the postcondition of an operation specification. The predicate is true, if such a message has been sent during the execution of *that* operation, and false otherwise.

If the Sieve class of our example were to define an *operation* `e`, we could specify the behavior of this operation as in the following example:

Example 2.1 The specification

```
context Sieve::e(z: Integer)
pre:  z > p
post: itsSieve->notEmpty() implies itsSieve^e(z)
```

means that if the received value `z` is greater than `p`, and the contextual object is associated to a successor, then it will send an `e` message to the successor with the value `z`.

Our example does not use any operation call, so we cannot say anything about the behavior of Sieve using current OCL. The intended behavior can, e.g., be specified by a state machine (see Fig. 3), but we want to specify the behavior of the object on a higher level of abstraction. We also want to separate the obtained specification from the object’s environment. This cannot be done with sequence diagrams.

Here we need a notation stating that the contextual instance has *received* a message. To order receiving and sending messages we also need *histories*.

To allow more complex reasoning about messages, OCL 2.0 introduces the message operator $\hat{\hat{}}$. The expression $\text{o}\hat{\hat{\text{msg}}}(\text{e})$ reads as “The sequence of all messages `msg` sent to an object `o` with parameter values `e` during the lifetime of the contextual object.” This operator projects on the history of all messages sent by the contextual object to a specific object with specific parameter values. We can use this operator to, e.g., require that the sequence of messages sent by an instance of Sieve is monotonically increasing.

Example 2.2 The expression

```
context Sieve
inv: let message : Sequence(OclMessage) =
    itsSieve^e(?: Integer) in
    Integer{2..message->size()}->forall(i |
        message->at(i).z > message->at(i-1).z)
```

intends to state that the sequence of `e` messages sent to the next sieve is monotonically increasing.

But how do we specify that the sequence of `e` messages *received* has to be

monotonically increasing? There is no language construct in OCL 2.0 which allows one to assert that an object has received a message during the execution of an operation or otherwise. Clearly, we cannot specify that, in order to send a message to another object, we need to have received a specific signal. There is no means to access the messages which we have received.

OCL has introduced some interesting notions for specifying behavior. But OCL 2.0 proposal allows only the specification of behavior by referring to sent messages in an operation's postcondition. We envisage that practice requires more expressive means to specify more general properties of a system's behavior, for example causality constraints.

3 Observables

We introduce a general formalism for specifying behavior in OCL. We prefer to reason in terms of sequences of observable events, or histories. This makes it possible to specify that invoking an operation is always preceded by receiving a signal. In this section we define our notion of an observable event in UML and OCL.

Messages represent signals or operations by their names and by the actual arguments sent. Events correspond to sending and receiving a message. To keep the presentation simple, we only consider the events of sending and receiving asynchronous signals, and invoking and returning from an operation. This notion of observable events can be refined further to take the event queue of active objects into account, to model object creation and synchronous signals. In our setting, events correspond to the source and the target of arrows in sequence diagrams.

We restrict our presentation to reliable communication, i.e., no message which is sent will get lost. Also, we assume an interleaving model for concurrency.

3.1 Events

Events drive the computation of UML models. An *event* is a specification of a kind of an observation, e.g., calling an operation or receiving a return value. The observation of an event is assumed to take place at an instant in time without duration. For our discussion we distinguish two kinds of events: signal events and call events.

An *asynchronous signal* models the asynchronous communication between two objects. The associated events are sending the signal and receiving the signal. Both kinds of events are represented using the data structure `OclEvent`.

A *synchronous signal*, formerly known as an *operation call*, models a synchronous communication between two objects. The associated events are sending the signal, receiving the signal, i.e., selecting it from the event queue, and *returning* from the reaction to the signal.

Events are different from actions and messages. An action may give rise to many events to be observed. The action of sending a signal allows one to observe that a signal is sent by one object or that the same signal is later received by another object.⁶

A message is a representation of a particular signal or call which is passed from one object to another. A message is part of many events. The same message may be observed to be sent or received.

3.1.1 Communication Record.

In this section we explain the basic structure describing the observation of an event, called *communication record*. Communication records are instances of the class `OclEvent` together with the `OclMessage` associated to the event, as shown in Fig. 5.

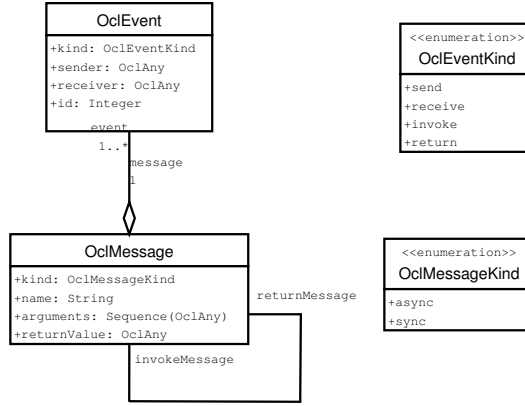


Fig. 5. Definition of an OCL Event

A message is shared between many instances of `OclEvent`, because the same message may take part in many observations. The following observations can be made when sending an asynchronous signal:

- (i) Sending a message representing the signal. This is represented by an instance of OCL event with a state of *send*.
- (ii) Receiving a message representing the signal. This is represented by an instance of OCL event with a state of *received*.

The message itself is unchanged by these events. This is represented by the multiplicity of 1 for the association from `OclEvent` to `OclMessage`.

A communication record stores the information relating to an observation. It has the following attributes:

- Its *sender*.
- Its *receiver*.

⁶ We avoid the complication of observing whether receiving a signal triggers a transition of a state machine, is discarded, or deferred.

- Its *message*. For an operation call we pass two messages: The first message is sent to initiate a call. It consists of the name of the operation and the actual parameter values passed to the receiver. The second message is used to acknowledge the completion of an operation call and to send return values back. For a signal we pass one message. A message consists of:
 - Its *kind*: Either *async* or *sync*.
 - Its *name*: The name of the operation or signal involved in this message.
 - Its *arguments*: A list of values representing the actual arguments sent with the message. A return message contains the actual parameter values of the invoking message along with the return value.
 - Its *return value*: This attribute holds the return value of an operation call.
 - Its *id*: This is a value used to uniquely identify a message. We assume a global counter which assigns to each message a unique integer in monotonically increasing order. This allows one, e.g., to specify overtaking of messages.
 - Its *event*: The events with which this message is associated. A message is generally associated with more than one event.
 - Its *invokeMessage*: If the message is a return message, the initiating invoke message can be accessed through this association.
 - Its *returnMessage*: If the message is a call message which has returned, then the return message can be accessed through this association.
- Its *kind*: One of the values from the life cycle. For signals this is either *send* or *received*, for operation calls this is either *invoke* or *return*.

Before we proceed with our presentation, we introduce some notation. Whenever a value of the communication record is *undefined*, we write \perp for this value. If an element e occurs in a sequence S , we write $e \in S$. If a sequence S is a *subsequence* of T , we write $S \sqsubseteq T$. We call an event *externally observable* if it can be observed from another object. This is the case whenever sender and receiver of a message are different. Messages sent from an object to itself are not observable.

We point out, that our `OclMessage` is different from the `OclMessage` of OCL 2.0 [3]. We discuss the relation between these data types in Sec. 3.3.

3.1.2 Asynchronous Signal Events.

We record signal events with one communication record in the sender's history and one communication record in the receiver's history. The value of the attribute *kind* of the communication record may be:

OclEventKind::send This value models that a signal was sent from the sender to the receiver, and appears only in the sender's history.

OclEventKind::receive This value models that the signal has been received by the receiver, and appears only in the receiver's history.

Example 3.1 Assume two objects called g and s . If g sends a signal called e with the actual parameter values n to s , we have the following records in their histories:

$$\langle \text{OclEventKind}::\text{send}, g, s, i, \langle \text{OclMessageKind}::\text{async}, e, \langle n \rangle, \perp \rangle \rangle \in g.\text{localHistory}$$

and

$$\langle \text{OclEventKind}::\text{receive}, g, s, j, \langle \text{OclMessageKind}::\text{async}, e, \langle n \rangle, \perp \rangle \rangle \in s.\text{localHistory} \quad .$$

Note that i and j are integers identifying the event such that $i < j$. We have omitted the values for *invokeMessage* and *returnMessage*. Both are undefined.

3.1.3 Synchronous Signal Events.

A synchronous signal event causes two synchronizations between the sender and receiver: First it synchronizes when the operation is invoked, and then it synchronizes when the operation completes and the return value is sent back. This is modeled by two *synchronous* messages sent between the two participating objects; the first message is used to invoke the operation, the other is used to send the return value back and to report completion of the operation call. Because both events are synchronous the communication records appear in both the sender's and receiver's history. The value of the kind attribute of the communication record may be:

OclEventKind::invoke The invoke record records the fact that the caller object calls an operation or method.

OclEventKind::return The return record models that the called object returns from an operation call of an operation or method.

Example 3.2 Assume two objects called o and p . If o calls an operation named m with the actual parameter values \mathbf{a} of the object p , which then returns the value v , we have the following subsequences of their histories:

$$\begin{aligned} & \langle \langle \text{OclEventKind}::\text{invoke}, o, p, i_0, \langle \text{OclMessageKind}::\text{sync}, m, \mathbf{a}, \perp \rangle \rangle, \\ & \quad \langle \text{OclEventKind}::\text{return}, p, o, i_3, \langle \text{OclMessageKind}::\text{sync}, m, \mathbf{a}, v \rangle \rangle \rangle \sqsubseteq \\ & \quad o.\text{localHistory} \end{aligned}$$

and

$$\begin{aligned} & \langle \langle \text{OclEventKind}::\text{invoke}, o, p, i_1, \langle \text{OclMessageKind}::\text{sync}, m, \mathbf{a}, \perp \rangle \rangle, \\ & \quad \langle \text{OclEventKind}::\text{return}, p, o, i_2, \langle \text{OclMessageKind}::\text{sync}, m, \mathbf{a}, v \rangle \rangle \rangle \sqsubseteq \\ & \quad p.\text{localHistory} \quad . \end{aligned}$$

Again, for any j the i_j are integers identifying the event such that $i_j < i_{j+1}$.

3.2 History

We have defined the events relating to the sending of signals and calling operations. A sequence of such events constitutes a history. Because a history is of type `Sequence(OclEvent)`, we can reuse the usually defined operations of `Sequence`. These operations are sufficient for most uses.

We distinguish between a *local* and a *global* history. A local history contains the externally observable events of a *single* object and describes the interface of it. A global history contains all observable events of the complete system. These observable events are all events generated by all objects of a system during its computation and the events received from its environment, in the order in which they occur.

The local history is introduced by adding to each object an attribute `localHistory` of type `Sequence(OclEvent)` to the class `OclAny`. A global history of the system is supplied in a similar manner using the attribute `globalHistory`. This is shown in Fig. 6:

```
context OclAny:
inv: localHistory->forall(e | (e.sender = self or
    e.receiver = self) and (e.sender <> self or
    e.receiver <> self))
inv: globalHistory->forall(e | e.sender <> e.receiver)
inv: OclAny->allInstances()->forall(o,p |
    o.globalHistory = p.globalHistory)
```

Fig. 6. Properties of Histories

3.3 Comparison to OCL 2.0

We show that our notion of history in OCL is at least as expressive as the message expressions proposed in OCL 2.0, proceeding as follows. First we show how the data types used for message expressions can be represented in our formalism. Then we explain how OCL 2.0 message expressions can be expressed as history expressions. We shall point out some semantic ambiguities in the standard, and how they may be corrected.

3.3.1 OclEvent and OCL 2.0's OclMessage

OCL 2.0 defines its own data type `OclMessage` [3, p. 6-4], which is syntactically different from ours. Here we explain the relation between both. The four operations on `OclMessage` defined in OCL 2.0 can be specified as follows:

```
context OclMessage::hasReturned(): Boolean
post: result = (kind = return or returnMessage->size() > 0)

context OclMessage::result(): T
pre: kind = return
post: result = returnValue
```

```
context isSignalSent(): Boolean
post: result = (kind = send)
```

```
context isOperationCall(): Boolean
post: result = (kind = invoke or kind = return)
```

Here T refers to the return type of the operation this message refers to.

3.3.2 The $\hat{}$ Operator

As described in Example 2.1 and in Section 2.7.1 on page 2-23 of [3], a message expression using the $\hat{}$ operator results in an instance of Boolean. It evaluates to true if and only if a message m with arguments \mathbf{a} has been sent during the execution of its contextual operation. To this end, we define the history of events sent and received during the execution of an operation, i.e., the sequence of all events between receiving an invoke and the corresponding return event:

```
let last: OclEvent = localHistory->at(localHistory->size()) in
let start: Integer =
  let search(i: Integer): Integer =
    if i >= 0 and not
      (localHistory->at(i).message.name =
        last.message.name and
        localHistory->at(i).message.arguments =
          last.message.arguments and
        localHistory->at(i).state = OclEventKind::invoke
        and
        localHistory->at(i).sender = last.receiver)
    then search(i - 1) else i endif
  in search(localHistory->size())
```

In a postcondition of an operation the value of `last` refers to the index of the event sending the return message back in the objects local history. `start` refers to the index of the corresponding invoke message of the operation call. Then `localHistory->subSequence(start, last)` is the sequence of all events observed during the execution of the contextual operation. Since we need only to check that the send event with the message m and with arguments \mathbf{a} occurs in this subsequence, the meaning of $\mathbf{o}^{\hat{}}\text{msg}(\mathbf{p})$ is:

```
localHistory->subSequence(
  start, localHistory->size())->select(m | m.state = send and
    m.receiver = o and
    m.message.name = m and
    m.message.arguments = a)->notEmpty()
```

By this we have established:

Proposition 3.3 *The OCL 2.0 expression $\mathbf{o}^{\hat{}}\text{msg}(\mathbf{e})$ is expressible as a local history expression.*

3.3.3 The $\hat{\hat{}}$ Operator

The exact semantics of the $\hat{\hat{}}$ operator is not precisely defined in the OCL standard; we base this discussion on the description of this operator on page 2-23 of the standard proposal [3] and on [13]. Recall, that the informal meaning of this operator is: “The sequence of all messages of a specific name sent to a specific object with some specific parameter values during the lifetime of the contextual object.” The standard proposal explains the $\hat{\hat{}}$ operator with:

```
context Subject::hasChanged() post: observer $\hat{\hat{}}$ update(12,14)
```

This results in the Sequence of messages sent. Each element of the collection is an instance of OclMessage.

It is not clear whether this expression refers to the sequence of messages sent during the life time of the contextual instance or only to the sequence of messages sent during the execution of the contextual operation. Because it is used in the same way as the $\hat{}$ operator, we assume the latter.

Then the semantics of the expression in our formalism is (analogous to Sec. 3.3.2):

```
localHistory->subSequence(  
  start, localHistory->size()->select(m | m.state = send and  
    m.receiver = o and  
    m.message.name = 'msg' and  
    m.message.arguments = e).message
```

Note the application of the navigation to message at the end of the constraint. This expression is not a sequence of instances of OclEvent but a sequence of instances of OclMessage, which (according to Sec. 3.3.1) is obtained as in the proposed standard. We note this result as:

Proposition 3.4 *The OCL 2.0 operator $o\hat{\hat{}}msg(e)$ can be expressed by a history expression.*

Proposition 3.3 and Proposition 3.4 imply that OCL 2.0 message expressions can also be expressed by history expressions.

Corollary 3.5 *History expressions are at least as expressive as OCL 2.0 message expressions.*

4 Local and Global Specifications

We have shown that, in addition to our data types and a mechanism which updates histories, everything needed for behavioral specifications is already present in standard OCL. In this section we show that our proposed extension is more general than the message expressions of OCL 2.0.

For our specification language we need quantification over sets and sequences, quantification over the set of all subsequences of a sequence, and projection operations applied to sequences.

- Quantification over collections are provided by the usual **forAll** and **exists** operations.
- Quantification over subsequences of a sequence can be expressed by using a function that computes the set of all subsequences of a sequence.
- Projection operations are expressed using **select** and **collect** operations.

We refine the concept of histories in OCL by the notion of local and global specifications. A *local* specification is a constraint on a *single* object. A *global* specification is a constraint on the links between a group of objects. This distinction is introduced to separate different concerns:

- Local specifications are used to specify the behavior of a single object in any possible environment in which this object can be used. Such specifications are used to constrain the instance variables of an object. Most important is that such a specification is not part of the interface of an object. A client of an object need not have any knowledge of these constraints.
- Global specifications are used to specify how different objects collaborate to achieve a common goal. In the global specification we constrain the environment of an object.

This separation of concerns is one of the fundamental contributions of component-based design. We consider each object also as a component. When considered as such, the signals it is able to receive and the operations it defines are the interface which this component *provides*. The interface it *requires* is made explicit by specifying which messages the object may send.

4.1 Local Specification Language

A local specification is a specification which refers only to the attributes of an object, the values of its associations, and its local history. Expressions which contain arbitrary navigations are not allowed as local specifications. It is only allowed to test association ends of an objects for equality or containment, whether such an association end is empty, or how many elements can be reached through it.

For our Sieve example, local specifications are

```
context Generator
```

```
inv: x > 1
```

```
context Sieve
```

```
inv: not oclInState(State_0) implies p > 1
```

```
inv: not oclInState(State_0) implies
```

```
  Integer{2..(p-1)}->forAll(i | p mod i <> 0)
```

Quantification is bounded by a local collection. A local collection is a collection which can be constructed without navigation expressions and the use of **allInstances**; e.g., **Integer{2..(p-1)}** is a local collection. We may construct such a collection from the local history, local attribute values, and

the local association relations:

```
context Sieve
inv: self <> itsSieve
inv: oclInState(State_1) implies itsSieve->notEmpty()
```

This restriction is imposed, because we want to make sure that the local specification refers only to the locally known object identities and the identities the object may have known in the past. The following is *not* a local constraint:

```
context Sieve
inv: oclInState(State_1) implies itsSieve.p > p
inv: oclInState(State_1) and itsSieve.oclInState(State_1)
    implies itsSieve.itsSieve->notEmpty()
```

The first invariant asserts that the object known to it as `itsSieve` has a value for `p` which is greater than its own. This is a statement about the other object, too.

Local specifications usually constrain the implementation of an object. As such, the client of such an object should not need to know about implementation details. For example, it is not necessary to know that the behavior of an object is implemented or specified by a state machine.

To describe the interface of an object we introduce local behavioral specifications. These are local specifications that only uses the local history of an object.⁷

The reason for this restriction is that any client of an object may only invoke operations or send signals specified in the object's interface. Then the client can at most observe the messages the object sends. The client cannot observe the state of an object, if this is not specified in the interface. Because the purpose of a local behavioral specification is to only specify its observable behavior there is no need to refer to the encapsulated state. Also, documenting and specifying the non-observable part of an object in its interface can be confusing for programmers, who want to use the object as a ready-made component. For example, a behavior of an instance of `Sieve` can be described as:

```
context Sieve
inv: Integer{1..localHistory->size()}->forall(i |
    localHistory->at(i).kind = send and
    localHistory->at(i).message.name = 'e'
    implies Integer{1..i}->exists(j |
        localHistory->at(j).kind = receive and
        localHistory->at(j).message.name = 'e' and
        localHistory->at(j).message.arguments->at(1) =
            localHistory->at(i).message.arguments->at(1)))
```

⁷ We consider all attributes as private. If we extend the event data type with events recording the reading and writing of other objects' attributes, then the attributes and association ends which are not declared private may also be used in the local specification language.

This means that an instance of Sieve only sends an `e` signal with a value if it has received one before it. This is a (weak) causality constraint. The observable behavior serves as an abstraction of the internal state. Given a history of events of an object we can simulate the object's behavior and therefore determine its internal state. All information necessary to do this is represented in this local history.

In the last example we do not refer to the sender of the signal received or the receiver of the signal sent. This cannot be done with state machines or message diagrams. It is, on the other hand, useful during top-down design, because we can postpone the decision of to whom we send the signal, or even decide to send it to self.

Another constraint on the history of our sieve example is, that each sieve object receives messages in increasing order. This can be expressed as:

```
context Sieve
inv: let recv = localHistory->select(e | e.kind = receive and
                                     e.name = 'e') in
    Integer{2..recv->size()}->forall(i |
        localHistory->at(i - 1).message.arguments->at(1) <
        localHistory->at(i).message.arguments->at(1))
```

4.2 Global Specification Language

Objects do not function alone. They interact with other objects. We specify how an object collaborates with its environment using a global specification language. This level of specification has two purposes:

- We can define global constraints on how the objects collaborate.
- We can give further constraints on how objects are linked together in a certain application. These global constraints are called component invariants in [2].

A global specification usually should not constrain an object's attributes, but it should constrain the links between objects, similar to architecture diagrams.

Example 4.1 Recall the sieve example. Then a global specification stating that the generator is associated to the sieve object with the smallest number is:

```
context Generator
inv: Sieve->allInstances()->forall(s | itsSieve.p <= s.p)
```

Global specifications may refer to the global history. For instance, stating formally that a message will be passed on to a different object is a global specification.

Example 4.2 Recall the Sieve example. One global property of the system is that if a number is not a prime it will eventually not be retransmitted.

```
context Generator
```

```

inv: Integer.allInstances()->forall(n | n>1 implies
  Sieve.allInstances()->exists(s |
    globalHistory->forall(e | (e.sender = s and
      e.kind = OclEventKind::send and e.message.argument->at(1)= n)
    implies Integer.allInstances()->exists(m |
      globalHistory->at(m) = e and
      Integer.allInstances()->forall(l |
        l > m implies
        globalHistory->at(l).message.arguments->at(1) <> n))))))

```

Unfortunately, OCL does not allow suitable abbreviations in this formula, but it reads: “For every integer n there exists an instance s of Sieve such that for each message e with argument n sent by s there exists a position m at which e occurs in the global history and for any position l after m in the global history the value n does not occur as an argument.”

5 Compatibility

Our purpose is to verify the feasibility of the composition of objects by proving consistency of a set of local behavioral specifications through the use of a *compatibility predicate*. Compatibility essentially means that a set of objects has a global computation. This means that for its local histories we can find a matching global history.

The main benefit of using a compatibility predicate is that it enables us to verify correctness properties of the system under development during the early stages of its design. For compatibility we only need

- (i) a specification of the globally desired behavior as specified by an invariant on the global history,
- (ii) a specification of how the objects of the system are composed and how they communicate, as specified by a global invariant, and
- (iii) the externally observable behavior of each object in the system as specified by an invariant on their local histories.

We do not need any further knowledge about the implementation of the objects in the system (e.g., about how those objects are composed themselves) and we can leave the specification of the objects’ implementation for later stages of the development process. Item (ii), for example, specifies that the asynchronous communication between the objects is in an first-in-first-out manner.

Essentially, the compatibility predicate establishes whether an n -tuple of local histories (χ_1, \dots, χ_n) of the participating objects o_1, \dots, o_n fit together in the sense that there exists a global history of their composition which, when projected on the composing object o_i , yields the original local history χ_i of the object one started out with.

Definition 5.1 Let O be a set of objects. To each $o \in O$ we assign a local history χ_o . We write χ for the list of local histories χ_o for $o \in O$ and $\text{proj}_o(\chi)$

for the projection of the history χ on the object o . Then the compatibility predicate is defined by:

$$\text{compat}(\chi) \leftrightarrow \exists \chi. \bigwedge_{o \in O} \text{proj}_o(\chi) = \chi_o \quad . \quad (1)$$

A UML model, however, does not specify the system in terms of objects but in terms of classes. We reformulate the compatibility predicate in terms of classes.

Given a class diagram consisting of classes C_1, \dots, C_n with associated local behavioral specifications $\varphi_1, \dots, \varphi_n$, and given a global invariant Φ on the object structure and the global history, we extend the definition of the compatibility predicate (1) as follows:

$$\exists \chi. \Phi \wedge \bigwedge_{i=1}^n \forall z_i. \varphi_i[z_i/\text{self}] \wedge z_i.\text{localHistory} = \text{proj}_{z_i}(\chi) \quad (2)$$

The variable χ denotes a global history, the expression $\varphi_i[z_i/\text{self}]$ denotes the result of substituting each occurrence of `self` by z_i , and z_i is an instance of C_i . This Definition (2) lifts a local property of an object to the global level.⁸

The predicate described in Eq. (2) can be expressed in OCL except for the existence of the global history χ . We use the name `globalHistory` as a Skolem-constant for it. The following expression describes the compatibility predicate in OCL:

```
context OclAny
-- The local histories are projections of the global one.
inv: globalHistory->select(e | e.sender = self or
    e.receiver = self) = localHistory
-- The local behavioral specification is an invariant of the
-- classes
```

The lifting of the local constraints to the global level is implicitly done in the semantics of OCL [19]. The global invariant can be formulated as an invariant of any class or “spread” among the classes.

To use the compatibility test it is sufficient to provide a constraint for the local history (if one is missing, we assume true, which is any history) and add the above constraint to `OclAny`. Besides writing suitable constraints on local histories, the compatibility test is easily implemented. Currently, we are working on a tool which enables the *verification* of the compatibility test using PVS [17].

We briefly sketch the soundness proof of our approach, assuming the correctness of every object and that each object is active, i.e., has its own thread

⁸ For technical convenience we assume that all attributes used in the local behavioral specifications are explicitly qualified with `self`. Otherwise, the substitution operation used later has to explicitly handle the use of unqualified attributes.

of control. The soundness proof is a generalization of the soundness proof of the compositional trace semantics of CSP-like processes communicating via channels to take into account the lifting of local specifications to the global level, and by this also to the level of objects, and that links between objects, which play the role of channels, may change and that different association ends may alias each other. See [9] for details.

6 Conclusions, Related Work, and Future Work

The message expressions discussed in this paper are introduced in OCL 2.0 [3]. In [13] Kleppe and Warmer define the semantics of the action clause in terms of histories of events. This paper inspired our definition of histories. We have shown that the proposal for message expressions in OCL fails to take received messages into account. We have introduced an extension of OCL which does take these received messages into account. Our formalism, based on histories of events, can emulate the message expressions of OCL 2.0 and is more expressive.

The idea of taking those messages which an object receives into account during specifying a system is commonly accepted in component-based design. Successful applications of this way of modeling are presented in [20], and are integrated into UML 2.0 [16].

An early extension of OCL with means to reason about received and sent events is [10]. The authors add some syntactic constructs which allow the modeler to specify that messages have been sent in OCL. Their extension lacks the possibility to state that two messages have been sent in a particular order, because they do not define a history. In [4] histories of events have been introduced and specifications are written in the observational μ -calculus. This extension of OCL allows one to reason about messages sent and received. But the authors do not allow the specifier to use the full power of their extension. The authors advocate a use of specification patterns. Experience with Bandera shows, that a library of patterns may become larger than the language on top of which they are build on. Closest to our extension is [6]. Here the authors introduce histories of time-stamped events and modal operators like *always* and *eventually*. Our formalism is lacking means to specify liveness properties. Since we extend our formalism to compositional specification of real-time systems, the only liveness property to require of such systems is progress of time (See [12] for references). All the extensions of OCL to sequences of events known to us do not consider compositional specifications, for which the specification language has to be *adapted* appropriately.

The verification method described in this paper is based on the adaptation of the definition of the compatibility predicate for communicating sequential processes [22].

Our extension of OCL is conservative. It does not add any new syntactic constructor to the language and does not lead to any change of the OCL

meta-model. Instead, histories and events are introduced as new data types and their operators can be defined within existing OCL. Using this approach makes the meaning of a message expression immediately apparent to users of OCL. The drawback is that such expressions are often hard to read.

We are currently investigating how we can automate the verification of models based on local and global specifications and the compatibility predicate. To do this, we plan to formulate the semantics of behavioral UML models in a theorem prover (for example PVS [17] or Isabelle/HOL [14]) and to translate OCL constraints into the same formalism as done in [5].

For certain settings it is even possible that compatibility can be checked without interaction of a user. One way to achieve this is the use of a tableaux method [21]. Tableaux methods allow the construction of counter-examples to a specification, as in model-checking techniques [8,18].

In the context of the IST project OMEGA (see www-omega.imag.fr) we are working on an extension of our formalisms to real-time systems. This extension will be similar to [11]. The authors of [11] also use histories of events define the semantics of real-time expressions.

References

- [1] America, P. and F. S. de Boer, *Reasoning about dynamically evolving process structures*, Formal Aspects of Computing **6** (1994), pp. 269–316.
- [2] Baumeister, H., R. Hennicker, A. Knapp and M. Wirsing, *OCL component invariants*, in: Luqi and M. Broy, editors, *Proc. Wsh. Monterey - Engineering Automation for Software Intensive System Integration*, U.S. Naval Postgraduate School, Monterey, 2001, pp. 208–215.
- [3] Boldsoft and Rational Software Corporation and IONA and Adaptive Ltd., “Response to the UML 2.0 OCL RfP (ad/2000-09-03),” (2003), revised Submission, Version 1.6 (ad/2003-01-07). Available for download at <http://www.omg.org/cgi-bin/doc?ad/2003-01-07>.
- [4] Bradfield, J., J. Küster Fillipe and P. Stevens, *Enriching OCL using observational mu-calculus*, in: R.-D. Kutsche and H. Weber, editors, *5th International Conference on Fundamental Approaches to Software Engineering (FASE 2002), April 2002, Grenoble, France*, number 2306 in LNCS (2002), pp. 203–217.
- [5] Brucker, A. D. and B. Wolff, *HOL-OCL: Experiences, consequences and design choices*, in: J.-M. Jézéquel, H. Hussman and S. Cook, editors, *UML 2002 - The Unified Modeling Language*, number 2460 in LNCS (2002), pp. 196–211.
- [6] Cengarle, M. V. and A. Knapp, *Towards OCL/RT*, in: L.-H. Eriksson and P. A. Lindsay, editors, *FME 2002: Formal Methods - Getting IT Right, International Symposium of Formal Methods Europe, Copenhagen, Denmark, July 22-24, 2002, Proceedings*, number 2391 in LNCS (2002), pp. 390–409.

- [7] Clark, T. and J. Warmer, editors, “Object Modelling with the OCL,” Springer-Verlag, 2002.
- [8] Clarke, E. M. and E. A. Emerson, *Design and synthesis of synchronization skeletons using branching time temporal logic*, in: *Workshop on Logics of Programs*, number 131 in LNCS (1981), pp. 52–71, published in 1982.
- [9] de Roever, W.-P., F. S. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel and J. Zwiers, “Concurrency Verification: Introduction to Compositional and Noncompositional Methods,” Number 54 in Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 2001.
- [10] D’Souza, D. F. and A. C. Wills, “Objects, Components, and Frameworks with UML: The CatalysisSM Approach,” The Addison Wesley object technology series, Addison Wesley Longman, Inc., 1998.
- [11] Flake, S. and W. Mueller, *An OCL extension for real-time constraints*, in: Clark and Warmer [7], pp. 150–171.
- [12] Hooman, J., “Specification and Compositional Verification of Real-Time Systems,” Number 558 in LNCS, Springer-Verlag, 1991.
- [13] Kleppe, A. and J. Warmer, *The semantics of the OCL action clause*, in: Clark and Warmer [7], pp. 213–227.
- [14] Nipkow, T., L. C. Paulson and M. Wenzel, “Isabelle/HOL – A Proof Assistant for Higher-Order Logic,” Number 2283 in LNCS, Springer-Verlag, 2002.
- [15] Object Management Group, “UML 2.0 Infrastructure Specification,” (2003), <http://www.omg.org/cgi-bin/doc?ptc/03-09-15>.
- [16] Object Management Group, “UML 2.0 Superstructure Specification,” (2003), <http://www.omg.org/cgi-bin/doc?ptc/03-08-02>.
- [17] Owre, S., N. Shankar, J. Rushby and D. Stringer-Calvert, “PVS System Guide version 2.4,” SRI International, Computer Science Laboratory, Menlo Park, CA (2001).
- [18] Queille, J. P. and J. Sifakis, *Specification and verification of concurrent systems in CESAR*, in: M. Dezani-Ciancaglini and M. Montanari, editors, *Proceedings of the 5th International Symposium on Programming*, number 137 in LNCS (1981), pp. 337–351.
- [19] Richters, M., “A Precise Approach to Validating UML Models and OCL Constraints,” Ph.D. thesis, Universität Bremen (2002), logos Verlag, Berlin, BISS Monographs, No. 14.
- [20] Selic, B., G. Gullekson and P. T. Ward, “Real-Time Object-Oriented Modeling,” John Wiley & Sons, Inc., New York, Chichester, Brisbane, Toronto, Singapore, 1994.
- [21] Smullyan, R., “First Order Logic,” Springer-Verlag, 1968.
- [22] Soundararajan, N., *Axiomatic semantics of communicating sequential processes*, ACM TOPLAS **6** (1984), pp. 647–662.