Verifying OCL specifications of UML models : tool support and compositionality

Kyas, M.

Chapter 5

Trace-based Compositional Specification and Verification for Object-Oriented Systems

The Object Constraint Language (OCL) is the established language for specifying properties of objects and object structures. OCL 2.0 has been extended with features that allow the specification of messages sent between objects.

We present a generalisation of this extension that allows to additionally specify causality constraints. From a pragmatic point of view, such causality constraints are needed to express, for example, that "each acknowledgement must be preceded by a matching request" in the context of exchanging messages, which is frequently required by communication protocols.

Our generalisation is based on the introduction of histories into OCL. Histories describe the external behaviour of objects. Moreover, to reason compositionally about the behaviour of a complex system we distinguish between *local* specifications within a single object and *global* specifications describing the interaction between objects. These two types of specifications are expressed in syntactically different dialects of OCL.

5.1 Introduction

OCL is used, among others, to constraining object structures in UML. Constraints on an object structure are invariants over a state of that structure. Such invariants use an object's attributes and the relationships between objects.

The behavioural concepts in OCL are pre- and postconditions of operations. The additional behavioural concept of a *message expression* has been introduced in OCL 2.0 [113]. Message expressions are predicates on traces of a particular object. As we shall demonstrate this imposes a real restriction.

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

OCL 2.0 introduces a history as a sequence of local snapshots which are part of the interpretation of an object's valuation [113, 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 behaviour within an object, and a *global* part describing the message exchange between different objects to achieve a common goal. More precisely, a local specification consists, as its first part, of a specification of the internal structure of an object. The second part of a local specification, which we call *local interface specification* or *behavioural specification*, provides a specification of the observable behaviour 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 introduced in this chapter 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 history is achieved. The compatibility predicate introduced in this chapter is a generalisation of the compatibility predicate for CSP described by Soundararajan in [144] for object-oriented systems. The verification step of checking the compatibility predicate relies only on the observable behaviour of objects 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 behaviour*. According to the encapsulation principle, a *behavioural* 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 support the decomposition in a natural way. Object-oriented design makes the interface of the parts of a large system explicit; aggregation, a kind of association, 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, as suggested by America [5] and Baumeister et al [9]:

1. A *local* specification is an OCL constraint on the local attributes and the local history of events of a single object, only.

2. A *local behavioural* specification is a local specification which only constrains the local history of an object and does not refer to the object's internal structure.

3. A *global* specification is an OCL constraint on the links, that is, references, *between* objects; a global specification constraints constrains the sequence of messages passed via these channels.

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

The remainder of this chapter is structured as follows: In the next section we summarise the extension of OCL 2.0 with message expressions and demonstrate a weakness of this extension. In Section 5.3 we describe the notion of events and histories used in our extension of OCL. We identify the *observable behaviour* of an object for the assertional specification of that object. We use the traditional choice of messages sent and received by an object. In Section 5.4 we describe how our method facilitates compositional specifications and reasoning. We elaborate on the distinction between local specifications, local behavioural specifications, and global specifications. In Section 5.5 we describe our compositional verification method. Finally, in Section 5.6 we draw some conclusions and compare our results to related work.

5.2 State of the Art and Motivation

We describe the means for specifying interaction between objects in OCL 2.0, and explain in what respect they are not sufficient from a pragmatic point of view. To explain this we use the Example of the *Sieve of Eratosthenes*, which we have introduced in Section 4.4, and follow the ideas presented by America in [5].

Whether the property that the value of each attribute p of an instance of Sieve is a prime number is satisfied by the model depends on the behaviour 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 reason for our extension.

OCL 2.0 introduces the two operators ^ and ^^ for reasoning about messages. The first one, $o\hat{}msg(e)$, is a predicate that 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 behaviour of this operation as in the following example:

Example 5.1. The specification

context *Sieve* :: $e(z : Integer)$
pre : $z > p$
post : $itsSieve \rightarrow notEmpty()\ implies\ itsSieve\hat{}e(z)$

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 behaviour of Sieve using current OCL. The intended behaviour can, for example, be specified by a state machine (see Figure 4.3), but we want to specify the behaviour of the object on a higher level of abstraction. We also want to separate the obtained specification from the object's environment.

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

To allow more complex reasoning about messages, OCL 2.0 introduces the message operator ^^. The expression $o\hat{}\hat{}msg(e)$ 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, for example, require that the sequence of messages sent by an instance of Sieve is monotonically increasing.

Example 5.2. The expression

context *Sieve*
inv : **let** $message : Sequence(OclMessage) = \quad itsSieve\hat{}\hat{}e(? : Integer)$ **in**
$Integer\{2..(message \rightarrow size())\} \rightarrow forAll(i \mid$
$message \rightarrow at(i).z > message \rightarrow at(i-1).z)$

intends to state that the sequence of messages sent to the next sieve consists of strictly monotonically increasing values. ♦

But how do we specify that the sequence of 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. Clearly, we cannot specify that, in order to send a message to another object, we need to have received a specific signal. There are no means in OCL to specify the messages which we have received.

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

5.3 Observables

We introduce a more general formalism for specifying behaviour in OCL. We prefer to reason in terms of sequences of observable events, called 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 or 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, that is, no message which is sent will get lost. Also, we assume an interleaving model for concurrency.

5.3.1 Events

Events drive the computation in UML models. An *event* is a specification of a kind of observation, for example, calling an operation or receiving a return value. The observation of an event is assumed to take place without duration at an instant in time. 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.

A *synchronous signal*, formerly known as an *operation call*, models a synchronous communication between two objects. The associated events are sending (and at the same time receiving) the signal and *returning* from the reaction to the signal.

All kinds of events are represented using the data structure *OclEvent*.

Events are different from actions, which may cause sending or receiving of messages, 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.[1]

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 both to be sent and to be received.

Communication Record.

In this section we explain the basic data structure which describes the observation of an event, called *communication record*. Communication records are instances of the class

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

OclEvent together with the *OclMessage* associated to the event, as shown in Figure 5.1.

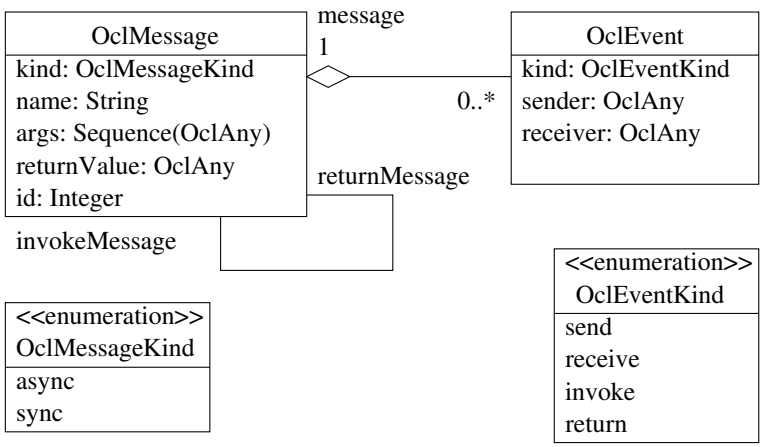

Figure 5.1: Definition of a communication record

An instance of *OclMessage* may be shared between many instances of *OclEvent*, because the same message may be part of many observations. On the other hand, each instance of *OclEvent* observes *exactly* one message. The enumeration OclMessageKind determines whether an instance of *OclMessage* refers to a message that has been sent asynchronously, indicated by the enumeration literal async, and thereby represents a signal, or whether it has been sent synchronously, indicated by the enumeration literal sync, and thereby represents an operation call. The following observations can be made when sending an asynchronous signal:

1. Sending a message representing the signal: This is represented by an instance of OCL event, where the attribute kind has the value of *send*.

2. Receiving a message representing the signal: This is represented by an instance of OCL event, where the attribute kind has the value of *receive*.

The case of synchronous messages, which represent operation calls, we observe two related events:

1. Invoking an operation by sending a message: This is represented by an instance of OCL event, where the attribute kind has the value of *invoke*.

2. Returning from an operation call by sending a return message: This is represented by an instance of OCL event, where the attribute kind has the value of *return*.

Send, receive, invoke, and return are literals of the enumeration *OclEventKind*. The multiplicity of 1 for the association from *OclEvent* to *OclMessage* models that each *OclEvent* refers to exactly one message.

A communication record, which is an instance of *OclEvent*, not only records the kind of the event, but also the information relating to its observation. It has the following attributes:

- Its *sender*.
- Its *receiver*.
- Its *message*: For an operation call two messages are sent: The first message is sent to initiate a call and consists of the operation's name and the actual parameters passed to the receiver. The second message is sent to indicate the completion of the call and to send return values back. For a signal only one message is sent.

  A message consists of:

  - Its *kind*, which is 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 expresses 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, for example, to specify overtaking of messages.
  - Its *event*: This lists 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* associated with the message. If the message represents a signal, the value is either *send* or *receive*. If the message represents an operation call, the value 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 $s \sqsubseteq t$.[2] We call an event *externally observable* if it can be observed

[2] A sequence s is a *subsequence* of t, if there exists a strictly monotonic function f such that $s_i = t_{f(i)}$ for all i.

from another object. This is the case whenever sender and receiver of a message are different. Messages sent by an object to itself are not observable.

We point out that our definition and the use of *OclMessage* is different from the *OclMessage* defined in OCL 2.0 [113]. We discuss the relation between these data types in Section 5.3.3.

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 5.3. 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:

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

and

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

Note that i and j are integer variables identifying the event such that $i < j$. We have omitted the values for *invokeMessage* and *returnMessage* which are both undefined. ♦

Operation Calls.

An operation call causes two synchronisations between the sender and receiver: First it synchronises when the operation is invoked, and then it synchronises when the operation completes and a return value is sent back. This is modelled by two *synchronous* messages: the first message is used to invoke the operation, the second is used to send the return value of the call back and to report completion of the operation call. The communication records appear in both the sender's and receiver's history, because they refer to synchronous communications . The value of the attribute kind of the communication record is:

OclEventKind::invoke This value models that a synchronous signal was sent by the sender and received by the receiver in order to invoke an operation.

OclEventKind::return This value models that a synchronous signal was sent by the sender and received by the receiver in order to indicate the completion of the operation call and to send back the return value.

Example 5.4. Assume the existence of two objects called o and p. If o calls an operation named m with the actual parameter values $\vec{a}$ of the object p, which then returns the value v, we have the following subsequences of their histories:

$$\langle\langle \mathit{OclEventKind} :: \text{invoke}, o, p, i_0, \langle \mathit{OclMessageKind} :: \text{sync}, m, \vec{a}, \bot\rangle\rangle,$$
$$\langle \mathit{OclEventKind} :: \text{return}, p, o, i_3, \langle \mathit{OclMessageKind} :: \text{sync}, m, \vec{a}, v\rangle\rangle\rangle \sqsubseteq$$
$$o.\mathit{localHistory}$$

and

$$\langle\langle \mathit{OclEventKind} :: \text{invoke}, o, p, i_1, \langle \mathit{OclMessageKind} :: \text{sync}, m, \vec{a}, \bot\rangle\rangle,$$
$$\langle \mathit{OclEventKind} :: \text{return}, p, o, i_2, \langle \mathit{OclMessageKind} :: \text{sync}, m, \vec{a}, v\rangle\rangle\rangle \sqsubseteq$$
$$p.\mathit{localHistory} \quad .$$

For any j the i_j express integers identifying the event such that for their values we have $i_j < i_{j+1}$. ♦

5.3.2 History

We have defined the events relating to sending and receiving of signals and calling operations. Sequences of such events are called *histories*. Histories are instances of *Sequence*(*OclEvent*). Therefore, we can reuse the operations defined for *Sequence* in the standard library (see Section 2.3.4). These operations are sufficient for almost all uses.

We distinguish between a *local* and a *global* history. A local history contains the externally observable events of a *single* object and describes its interface. 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 into OCL by extending the class *OclAny* with the attribute *localHistory* of type *Sequence*(*OclEvent*). Recall that *OclAny* is the (implicit) base class of all other classes, see Section 2.3.4, which implies that *all* objects inherit the attribute *localHistory*. A global history of the system is supplied in a similar manner using the attribute *globalHistory*. This is shown in Figure 5.2.

5.3.3 Comparison to OCL 2.0

We argue that our notion of history in OCL is at least as expressive as the message expressions proposed in OCL 2.0. First we show how the data types used for mes-

context *OclAny*
inv : *localHistory* → *forAll*(*e* | (*e*. sender = *self or e*. receiver = *self*) *and*
(*e*. sender <> *e*. receiver)
inv : *globalHistory* → *forAll*(*e* | *e*. sender <> *e*. receiver)
inv : *OclAny.allInstances*() → *forAll*(*o*, *p* | *o.globalHistory* = *p.globalHistory*)

Figure 5.2: Properties of histories

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

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

context *OclMessage* :: isSignalSent() : *Boolean*
post : *result* = (kind = sent)

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

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

Figure 5.3: Properties of OCL 2.0's *OclMessage*

sage 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.

OclEvent and OCL 2.0's OclMessage

OCL 2.0 defines its own data type *OclMessage* [113, p. 6-4], which is syntactically different from ours. Here we explain their relation. The four operations on *OclMessage* defined in OCL 2.0 can be specified as in Figure 5.3.

The ^ Operator

As described in Example 5.1 and in Section 2.7.1 on page 2-23 of [113], a message expression using the ^ operator results in an instance of Boolean. It evaluates to true if and only if a message m with arguments $\vec{a}$ has been sent during the execution of its contextual operation. To this end, we define the history of events sent and received dur-

ing the execution of an operation, that is, 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* =
        localHistory → *at*(last).*message.name*
      and localHistory → *at*(*i*).*message*.args =
        localHistory → *at*(last).*message*.args
      and localHistory → *at*(*i*).kind = *OclEventKind* :: *invoke*
      and localHistory → *at*(*i*).sender = *localHistory* → *at*(last).receiver)
    then search(*i* − 1) **else** *i* **endif**
  in search(*localHistory* → *size*())

In a postcondition of an operation last refers to the index of the event sending the return message back to the caller in the callee's local history, whereas start refers to the index of the corresponding invoke message of this operation call. Then *localHistory* → *subSequence*(start, last) expresses 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 $\vec{a}$ occurs in this subsequence, the meaning of *o*^*msg*(*e*) is:

localHistory → *subSequence*(start, *localHistory* → *size*()) →
  select(*m* | *m.state* = *send and m.receiver* = *o and*
    m.message.name = 'msg' *and m.message.args* = *e*) → *notEmpty*()

By this we have established:

Proposition 5.1. *The OCL 2.0 expression o^msg(e) is expressible as a local history expression.*

The ^^ Operator

The exact semantics of the ^^ 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 [113] and in the paper by Kleppe and Warmer [78]. 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 provides the following example as an explanation of the ^^ operator:

> **context** Subject :: hasChanged() **post** : observer^^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 ^ operator, we assume the latter.

The semantics of the expression in our formalism is (analogous to Section 5.3.3):

$$\begin{aligned} &\mathit{localHistory} \rightarrow \mathit{subSequence}(\text{start}, \mathit{localHistory} \rightarrow \mathit{size}()) \rightarrow \\ &\quad \mathit{select}(m \mid m.\mathit{state} = \mathit{send}\ \mathit{and}\ m.\mathit{receiver} = o\ \mathit{and} \\ &\qquad m.\mathit{message}.\mathit{name} = \text{`msg'}\ \mathit{and}\ m.\mathit{message}.\mathit{args} = e).\mathit{message} \end{aligned}$$

Note the application of the navigation operator 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 Section 5.3.3) is obtained as in the proposed standard. We note this result as:

Proposition 5.2. *The OCL 2.0 operator o^^msg(e) can be expressed by a history expression.*

Proposition 5.1 and Proposition 5.2 imply that OCL 2.0 message expressions can also be expressed by history expressions.

Corollary 5.3. *History expressions are at least as expressive as OCL 2.0 message expressions.*

5.4 Local and Global Specifications

We have shown that, in addition to our data types and a mechanism which updates histories, everything needed for behavioural 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 the internal structure of a *single* object. A *local behavioural* specification is a constraint on the externally observable behaviour of a single object, expressed as a constraint on its local history. A *global* specification is a constraint on the links, which describe whether one object can send a message to another object, between a group of objects, and the values communicated along these links. This distinction is introduced to separate different concerns:

- Local specifications are used to specify the behaviour 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, because the constrained attributes are not part of the interface. A client of an object need not have any knowledge of these constraints.

- Local behavioural specifications are constraints on the local history of an object.

  Observe, that local specifications are *not* constraints on local histories.

- Global specifications are used to specify the relations between different objects, that is, they constrain the values of associations, and 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*. By specifying which messages an object *sends*, the interface it *requires* is made explicit, because it describes a requirement on the messages the *receiver* has to provide.

5.4.1 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 navigation operators 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*(s_0) *implies* $p > 1$
inv : *not oclInState*(s_0) *implies Integer*$\{2..(p-1)\} \rightarrow$ *forAll*($i \mid p \bmod i \neq 0$)

Quantification is bounded by local collections. A local collection is only constructed from an OCL expression which do not contain navigation and *allInstances* expressions; for example, *Integer*$\{2..(p-1)\}$ expresses 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* $\neq$ *itsSieve*
inv : *oclInState*(s_1) *implies itsSieve* $\rightarrow$ *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*(s_1) *implies itsSieve.p* $> p$
inv : *oclInState*(s_1) *implies* (*itsSieve.oclInState*(s_1)
implies itsSieve.itsSieve $\rightarrow$ *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 behaviour of an object is implemented or specified by a state machine.

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

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 behavioural specification is to only specify its observable behaviour 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 behaviour of an instance of Sieve can be

described by:

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))))

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.[3] The observable behaviour serves as an abstraction of the internal state. Given a history of events of an object we can simulate the object's behaviour and therefore determine its internal state, once we know its initial state. All information necessary to do this is represented in this local history.

In our previous 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.receive and e.name* = ‘e’) **in**
Integer{2..recv → *size*()} → *forAll*(*i* | *localHistory* → *at*(*i* − 1).*message.args*
→ *at*(1) < *localHistory* → *at*(*i*).*message.args* → *at*(1))

5.4.2 Global Specification Language

Objects do not function alone. They usually 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 by Baumeister et al in [9].

[3]The constraint is weak, because it allows that one send observation may cause an arbitrary number of receive observations. The constraint can be strengthened to require that for each receive we have exactly one receive. See Section 6.5.2 for examples of how this may be specified.

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

Example 5.5. Recall the sieve example. Then a global specification stating that the generator is associated to the sieve object with the smallest number is:

$$\textbf{context } \mathit{Generator} \textbf{ inv} : \mathit{Sieve} \to \mathit{allInstances}() \to \mathit{forAll}(s \mid \mathit{itsSieve}.p \leq s.p)$$

⧫

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 5.6. 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.

$$\begin{array}{l}
\textbf{context } \mathit{Generator} \\
\textbf{inv} : \mathit{Integer}.\mathit{allInstances}() \to \mathit{forAll}(n \mid n > 1 \textit{ implies} \\
\quad \mathit{Sieve}.\mathit{allInstances}() \to \mathit{exists}(s \mid \mathit{globalHistory} \to \mathit{forAll}(e \mid \\
\qquad (e.\mathit{sender} = s \textit{ and} \\
\qquad\quad e.\mathit{kind} = \mathit{OclEventKind} :: \text{send } \textit{and } e.\mathit{message}.\mathit{args} \to \mathit{at}(1) = n) \\
\qquad \textit{implies } \mathit{Integer}.\mathit{allInstances}() \to \mathit{exists}(m \mid \mathit{globalHistory} \to \mathit{at}(m) = e \textit{ and} \\
\qquad\quad \mathit{Integer}.\mathit{allInstances}() \to \mathit{forAll}(\ell \mid \ell > m \textit{ implies } \mathit{globalHistory} \to \\
\qquad\qquad \mathit{at}(\ell).\mathit{message}.\mathit{args} \to \mathit{at}(1) \neq n)))))
\end{array}$$

Unfortunately, OCL does not allow suitable abbreviations in this formula. 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." ⧫

5.5 Compatibility

Our purpose is to verify the feasibility of the composition of objects by proving consistency of a set of local behavioural 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:

1. A specification of the globally desired behaviour as specified by an invariant on the global history,
2. a specification of how the objects of the system are composed and how they communicate, as specified by a global invariant, and

3. the externally observable behaviour 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 (for example, about how those objects are composed themselves) and we can leave the specification of the implementation of these objects for later stages of the development process.

Essentially, the compatibility predicate establishes whether an n-tuple of local histories $(\chi_1, \ldots, \chi_n)$ of the participating objects $o_1, \ldots, o_n$ 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 5.4. Let O be a set of objects. To each $o \in O$ we assign a local history χ_o. We write $\vec{\chi}$ 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:

$$\text{compat}(\vec{\chi}) \stackrel{\text{def}}{=} \exists \chi . \bigwedge_{o \in O} \text{proj}_o(\chi) = \chi_o \quad . \tag{5.1}$$

◊

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 $C_1, \ldots, C_n$ with associated local behavioural specifications $\varphi_1, \ldots, \varphi_n$, and given a global invariant Φ on the object structure and the global history, we extend the definition of the compatibility predicate (5.1) as follows:

$$\exists \chi . \Phi \wedge \bigwedge_{i=1}^{n} \forall z_i . \varphi_i[z_i/self] \wedge z_i.localHistory = \text{proj}_{z_i}(\chi) \tag{5.2}$$

In this expression, the variable χ denotes a global history, the expression $\varphi_i[z_i/self]$ denotes the result of substituting each occurrence of *self* by z_i, and z_i is an instance of C_i. This Definition (5.2) lifts a local property of an object to the global level.[4]

The predicate described in Equation (5.2) 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*
inv : globalHistory → *select*(*e* | *e*. sender = *self or e*. receiver = *self*) = localHistory

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

The lifting of the local constraints to the global level is implicitly done in the semantics of OCL provided by Richters [133]. 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. The tool described in Chapter 4 has been extended to handle these trace specifications which enables the *verification* of the compatibility test using PVS.

5.6 Conclusions, Related Work, and Future Work

The message expressions discussed in this paper are introduced in OCL 2.0 [113]. Kleppe and Warmer define the semantics of the action clause in terms of histories of events [78]. 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 modelling are presented by Selic et al in [142], and have been integrated into UML 2.0 [111].

An early extension of OCL with means to reason about received and sent events is presented in the Catalysis Approach by D'Souza and Wills [49]. D'Souza and Wills add some syntactic constructs which allow the modeller 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. Bradfield, Küster Filipe, and Stevens introduce histories of events into OCL and specifications are written in the observational μ-calculus [17]. 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.

Very similar to our extension is the one by Cengarle and Knapp [22]. They introduce histories of time-stamped events and modal operators like *always* and *eventually*. Our formalism is lacking means to specify liveness properties and only permits to specify of safety properties.

A very similar formalisation of OCL message expressions has been independently developed by Flake and Mueller [54]. While they define the semantics of OCL message expressions in terms of sequences of observations, similar to our local histories, they

do not introduce the concept of a history into the OCL. Consequently, they are not able to reason about received messages. However, they indicate that there is a need for temporal extensions of OCL.

Since we extend our formalism to compositional specification of real-time systems in Chapter 7, the only liveness property to require for real-time systems is progress of time (see Hooman [64] 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 [144].

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.

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 [143]. Tableaux methods allow the construction of counter-examples to a specification, as in model-checking techniques [28, 130].