On the Validation of Invariants at Runtime

Piotr Kosiuczenko
Institute of Information Systems, WAT
Kaliskiego 2, 00-908 Warszawa, Poland
E-mail: pkosiuczenko-NA-wat.edu.pl

Abstract: The paradigm of contractual specification provides a transparent way of specifying object-oriented systems. In this paradigm, system consistency is specified using so-called invariants. Runtime validation of invariants is a complex problem. Existing validation methods require either exhausting validation of an invariant for all objects of the corresponding class or the use of restrictive type systems. In this paper a non-exhaustive method of invariant validation is proposed. It is proved that the method is sound and an implementation of this method is discussed. It is shown also how to extract context free languages corresponding to OCL-invariants.

Keywords: OCL, design by contract, invariant validation, framing, optimization

1. Introduction

Contractual specification is the prevailing way of specifying object-oriented systems (cf. [32, 31]). A number of contractual languages exist, e.g., Object Constraint Language (OCL [38, 43]) related to UML, Spec# [8] related to C# and Java Modeling Language (JML [29]) related to Java. A contract consists of three basic constraint types: invariants, pre- and post-conditions. A pre-condition specifies when a method can be called and a post-condition specifies the system state after the method execution. The system consistency is ensured by invariants. Basically, invariants must hold in all visible system states, i.e., states preceding and following public method calls. However, it should be noted that there are various definitions of invariants’ validity (cf. e.g. [34]).

The validity of contracts can be monitored during system execution. The case of pre- and post-conditions is less problematic than the monitoring of invariants since pre- and post-conditions can be checked for actual method parameters without the need of inspecting all objects of a certain class (unless primitives like the OCL feature allInstances are used). If an invariant concerning objects

This is a technical version of the paper: On the Validation of Invariants at Runtime, accepted for publication in Fundamenta Informaticae, IOS Press, 2013
of a class $C$ includes an attribute of another class, then any modification of this attribute can potentially make this invariant invalid; even if the executed method does not primarily concern objects of class $C$. In general a method’s execution can make invariants corresponding to different classes invalid. An invariant corresponding to a class $C$ with one variable concerns all objects of this class. Therefore, if a brute-force validation method is applied, then the invariant has to be checked for all objects of that class. If an invariant has more than one variable, then it has to be checked for all combination of arguments; this may be computationally very expensive.

The runtime validation of invariants has highly polynomial time complexity. If an invariant has $k$ variables which range over objects of class $C$, checking this invariant requires $n$ steps and class $C$ has $m$ elements, then checking its validity requires $m^k \cdot n$ steps. To make sure that an invariant is satisfied in all visible states, it must be checked whenever an observable state is entered. This procedure must be performed for all invariants. Often one method call results in a number of other method calls, so $m^k \cdot n$ has to be multiplied by that number. The time needed to validate an invariant may depend on the number of objects in a class so that the time complexity of the validation is even higher. All this can cause unacceptable slowdown of method’s execution even in the testing phase.

To minimize the time complexity in some approaches invariants are checked only when a method returns, but not when it is called. However, as pointed out by Barnett et al. [8], this is an incorrect approach; re-entrance makes it impossible to treat object invariants as implicit pre- and post-conditions and to hide those pre-conditions from callers. Usually when a new visible state is entered, most invariants do not need to be re-evaluated and there are only few objects at which an invariant is made invalid, if at all. Thus, we need a method to decide which invariants needs to be re-evaluated and for which objects.

There were some works done to minimize the number of invariants checks [41] (see also section 2). The method is based on paths navigated when an invariant is evaluated. However, as pointed out by Briand et al. [9], finding navigation paths even in the case of simple OCL formulas is a complicated process even if done on the case by case basis. Their definition is even more difficult if recursion is used.

Banerjee et al. [7] proposed the notion of formula footprint, i.e., a minimal set of objects and attributes determining its value. The goal was, among others, to specify when the program cannot make a formula invalid. It is based on the concept of disjoint heap parts developed in the realm of Separation Logic [37]. The concept of regions, as defined in the paper [7], requires a more expressive language than JML. Regions can be defined by so called dynamic frames, i.e., specification variables whose values are sets of allocated objects [24]. JML does not include complex navigational and reachability formulas nor inductive definitions since the idea is to specify and prove weaker properties using a Hoare-style logic (cf. [7], p. 388). OCL is a perfect means to define regions and specification variables since it is a navigational languages including recursive definitions and a simple form of set theory, in particular selection operations and quantifiers. The high expressivity and abstractness of OCL comes at a price. It is hard to implement the validation of OCL-constraints. However, it is possible to automatically translate OCL-constraints to JML [5].

In this paper we investigate how the run-time validation of OCL-invariants can be done without the need to search all existing objects. We demonstrate that in some cases the search space can be significantly reduced. Our method allows us to identify all invariants which need to be re-evaluated
when an observable state is entered. We present a method of extracting context free languages from OCL-terms, and in particular from OCL-invariants. Extracted languages define the object networks which are navigated when an invariant is evaluated, and consequently frame the invariant. An object for which an invariant is evaluated can be treated as a root of such a network. If an invariant is violated by a method execution, then the corresponding network of objects is modified. We use extracted languages to navigate backwards from the modified places to the roots of invariants’ networks.

Since OCL is a very extensive language, we restrict our consideration to some core primitives and we prove our results for the restricted form of this language. Nevertheless, our results can be applied to a wide variety of invariants. Such invariants can include multiple variables, object- as well as class attributes, the allInstances feature and also recursively defined functions. The presented method allows us to identify invariants which have to be evaluated before and after method execution, and to identify objects for which the evaluation should be performed. It should be noted that our method does not depend on a particular definition of visible system states and therefore can be used with different definitions. The interesting thing is that we do not have to refer to all objects existing in the pre-state, but only to the roots of modified networks. Those roots can be defined by OCL-terms. We briefly discuss the issue of implementation, though it goes beyond the scope of this paper.

This paper is organized as follows. The next section includes a short discussion of related work. In section 3, we discuss the problem of invariant validation, sketch our solution ideas and present two examples. In section 4, we show how to extract context free languages from OCL-terms and investigate the relation between such languages and OCL-terms. We also prove that terms formalizing words are well typed. In section 5, we prove that the problem of invariant validity can be reduced to the problem of preservation of object networks determined by languages corresponding to invariants; we show also how to express such languages using OCL-terms. In section 6, we show how the developed methods can be applied to the examples introduced in section 3. In section 7, we briefly discuss the issue of implementation. Section 8 concludes this paper.

2. Related Work

The concept of pre- and post-conditions goes back to the idea of Hoare logic (cf. [21]). The concept of invariants can be traced back to the Hoare’s paper on data representation [22]. For the first time, contracts were used in Eiffel [32]. Java Modeling Language (JML) is a contractual languages corresponding to Java (cf. [29]) and Spec# corresponds to C# (cf. [8]).

There are various definitions of invariant validity. The most basic one says that invariants must be true when a public method is either called or terminates (cf. e.g. [10, 29]). In the case of the Object Constraint Language, invariants specify what must be true for each instance at all times, except when an object is executing an operation (cf. [38]). The newer version of JML restricts the definition to states proceeding and following calls to so-called nonhelper methods (cf. [29]). Spec# uses the notion of visible object states; an object is visible if it is currently not modified [8]. An invariant must hold in all visible objects’ states (cf. [34] for alternative definitions).

Validation of constraints during program execution requires efficient algorithms and proper tool support. There exist numerous tools allowing for OCL constraint validation, and discussing all of
them would go far beyond the scope of this paper. For example, there exist Use (cf. [15]), EOS (cf. [12]), KeY (cf. [3]) and Dresdener OCL Toolkit (DOT [13]). We refer the interested reader to the papers [6, 9, 14] for a tool overview. Clavel et al. [12] compare different tools with respect to the efficiency of OCL constraint evaluation. Some tools attempt optimization of invariant validation. For example, in the case of DOT, instead of checking all invariants before and after every public operation, invariants are checked after methods that modify attributes used in invariants. This is not easy to implement. DOT introduces a mirror attribute for every attribute in every class, regardless if the attribute is used in an invariant or not. There are also mechanisms to detect attribute value changes and to identify invariants which have to be checked (see the papers [9, 14] for an overview of invariant validation methods used in OCL tools).

A general method for the invariant checking optimization has been proposed by Van Der Straeten et al. [41], and followed in a number of tools including DOT. The authors classify invariants on the basis of paths navigated during invariant evaluation. The invariants, when found are checked for all existing objects. It is demonstrated how AspectJ can be used to implement the checking of identified types of invariants. However, they do not present any formal extraction algorithm allowing an identification of those paths. Proposed method does not allow one to restrict the number of objects for which an invariant has to be evaluated. There was some work done on optimization of constraint validation by rewriting and decomposing the constraints, and also on the minimization of the number of navigation steps needed to validate an invariant (see the paper [33] and the references there). Gopinathan et al. [17] show how to identify commands in a program source code which violated an invariant during an execution.

Modularity of specification and verification techniques is one of the most desired properties, as pointed out by Leino [30]. Optimally, one needs a methodology for a compositional system specification allowing an independent class specification and their composition. The problem of invariant monitoring and modularity can be circumvented by the concept of component and ownership. The idea is to restrict invariants to objects within a component and to restrict access to those objects from the outside of a component. One can use a type system, such as Universe in JML, which guarantees that objects in a component are uniquely possessed by the owning component. This type system controls the objects’ aliasing and disallows disclosing of the aliases outside of the owning component (see the papers [35, 36] and the references there). In this approach, objects are grouped in tree like structures where the parents have total control over their children. It permits modular invariant specification, but imposes strict restrictions on the system structure. In case of Spec#, there is a concept of ownership as well, but the ownership constraints must be satisfied only at visible objects’ states, as pointed out by Barnett et al. [8]. When an object’s state is invisible, the ownership can be passed from one owner to another. In both approaches, an invariant of class A may concern only objects within components owned by objects of that class. Such an invariant may include at most one free variable ranging over this class. In the current version of Spec#, class attributes cannot occur in invariants (cf. [8]).

The primary focus of JML and Spec# is on modular program specification and verification as well as run-time validation. The situation is different in case of OCL. OCL is a very powerful, high level specification language with the primary focus on expressivity rather than modularity, theorem proving or validation. Invariants can include an arbitrary number of variables, attributes, in particular
class attributes, and association-ends. They can also include the predefined feature allInstances referring to all objects of a class. OCL is good for specifying client-server systems. The general concept of global invariants, pre- and post-conditions fits well to this architectural paradigm. It can also be used to specify more general systems, though their main focus is on states before and after method’s execution. On the other hand, it is less suited for a modular system specification.

In this paper we do not focus on the complexity of the proposed method, but we provide its rough estimate to make clear that a systematic treatment of invariant validation is needed. However, it should be noted that classical complexity measures are a bit inadequate in this case as the complexity may depend on the number of objects of a particular class and their interrelationships which vary during program execution. Other models of complexity can be used here (cf. e.g. [39, 19]).

3. The Problem of Invariant Validation

In this section we discuss the problem of runtime invariant validation and informally outline the solution. We also present two simple examples in order to explain the problems with invariant validation. In the first one we specify an airline system and explain the problem of invariant specification. In the second one we specify a list storing reals. We use it in the following sections as the running example.

3.1. Run-Time Invariant Validation

Invariants specify consistency criteria and concern all existing objects of constrained classes. They should be preserved by method invocations. The validity of an invariant usually depends on a network of objects. More precisely, let \( I(\text{self}) \) be an invariant such that \text{self} ranges over objects of class \( C \). Even for a single object, the validity of \( I \) may depend not only on the state of \text{self} but also on states of various objects from different classes. To evaluate the invariant it is necessary to navigate through such networks of objects. A set of objects and attributes determining the value of \( I \) is called frame of \( I \) and its minimal subsets are called footprints [7]. A modification of a frame can cause the invariant to be invalid. Methods from classes different from \( C \) can modify footprints of \( I \) and there is no control over the preservation of \( I \). Moreover, they can modify footprints of different invariants. The validity of invariants has to be checked before and after a method execution. A brute force approach requires the validation of \( I(\text{self}) \) for all objects of class \( C \) and similarly for all other invariants. This has to be done for each method invocation. This may cause an unacceptable method slowdown.

The aim of this paper is to define a method of invariant validation at runtime which would avoid checking all invariants for all objects. The idea is first to statically define regular languages corresponding to paths navigated during an invariant validation. Thus, for invariant \( I(\text{self}) \), we define a regular, prefix-closed language \( L_{\text{self}}(I) \). The terminals occurring in words of \( L_{\text{self}}(I) \) correspond to attributes and associations occurring in the invariant. Those words correspond to paths navigated during validation of \( I \). In fact, such paths determine the validity of the invariant, and consequently frame \( I \).

The next step is to define a regular language corresponding to paths leading to modified attributes. For an object attribute \( a \), the corresponding regular language has the form \( L_{\text{self}}(I)/a \), i.e., it includes
all words $w$ such that $w a \in L_{\text{self}}(I)$. We can reverse the language $L_{\text{self}}(I)/a$. The reversed language determines the reversed paths leading from $a$ to self. We show that the reversed language is context free as well. We show that the objects navigable backward from $a$ using the reversed paths can be expressed by an OCL-term $s_{\text{self} I_0/a}^\text{rev}$. By definition, $s_{\text{self} I_0/a}^\text{rev}$ includes self. Thus, after a modification of $a$, we can restrict the validation of $I$ to $s_{\text{self} I_0/a}^\text{rev}$. We show that $s_{\text{self} I_0/a}^\text{rev}$ is well typed. We argue that in case when a class attribute is modified, we have to evaluate $I$ for all objects of class $C$. It should be noted that our results concern terms with an arbitrary number of variables.

3.2. Airline Example

In this subsection we present an example of an OCL-specification illustrating the basic ideas of our approach. OCL is a very expressive navigational language based on a simple form of set theory and relational algebra [38]. It is used in the context of UML class diagrams to give them a precise meaning. The types of OCL-expressions are defined relatively to such a diagram. The class diagram in Figure 1 includes class Airline associated with class Pilot. Being a pilot is a role. Therefore, class Pilot extends class Role. Class Person is associated with class Role via association-end role. Class Taxman is associated with class Person. We augment the diagram in Figure 1 with OCL-invariants and method specifications. Invariants specify consistency conditions. Pre- and post-conditions specify what must be true before and after a method execution. OCL-constraints are always defined in the context of a selected class from a class model. Such a class is indicated by the keyword context.

States of this system have to satisfy the following two invariants (we use iff as an abbreviation for two implications):

```
context Taxman inv tax_inv :
    self.people->forAll(p : Person | p.highEarnings iff p.role.salary->sum() >= 2000)
context Airline inv airline_inv :
```
**The first invariant is called** tax\_inv **and constraints all existing objects of class Taxman. They are ranged over by variable self. All objects of class Person related with self via association people must have attribute highEarnings set to true, and only if, the salaries for all the roles they play sum up to a number larger than or equal to 2000. It should be noted that self.people is a set of objects of class Person, not a single object. The general quantifier forAll ranges over elements of this set. The navigation in OCL is possible via collections rather than single elements. In the simple case, they are sets, but they can be also multisets and sequences [38]. For the sake of simplicity, we consider in this paper only navigation via sets.**

The second invariant restricts the salaries of pilots working for an airline. The method raise modifies the attribute salary of the actual implicit parameter by adding amount s. The invariability clause modifies : self::salary disallows any other change of objects which exist before the method execution, but it does not disallow the creation of new objects [26]. The post-condition of raise does not guarantee the preservation of tax\_inv and airline\_inv. It may happen that the method is applied to an object indirectly constrained by these invariants, i.e., being a part of their footprint. In such a case, it is hard to figure out for which objects the invariant could be violated, unless those invariants are evaluated for all existing objects. As mentioned in the introduction, JML [29] and Spec\# [8] deal with this problem by defining object hierarchies. An owning object controls its component objects entirely and only the owner has access to its components; in particular no direct method calls to owned objects can be made by objects different from the owner. A JML invariant for a class can concern only attributes and components of objects of that class. Since OCL is a general specification language, one can formulate arbitrary invariants, such as those ones specified above. Those invariants are inexpressible in Spec\# and JML since it is not possible to define an ownership relation guaranteeing an exclusive ownership of Role objects and since the invariants tax\_inv and airline\_inv refer to the same Role objects.

We apply the idea sketched in the previous section to this specification. We have to identify objects of class Taxman for which the corresponding invariant may be violated. We denote the first invariant by I_{tax\_inv}. First we define the language \( L_{self}(I_{tax\_inv}) \) corresponding to paths leading from objects of those classes to modified attributes. It defines paths labeled with people highEarnings, people role salary and their prefixes. They determine the validity of I_{tax\_inv}. Division by salary yields the path people role. We can navigate back to roots of modified networks via reversed associations role^{rev} and then people^{rev}. Thus, if attribute salary is modified for object self, then the OCL-term self.role^{rev}.people^{rev} specifies objects of class Taxman for which the invariant has to be checked. In case of the second invariant, first we have to select objects from class Role which in fact belong to class Pilot, downcast them and then apply the reversed association pilots^{rev}. 
3.3. List

In this subsection we present a list example. This will be the running example to explain problems with invariant validation in case of recursively defined structures.

![List Diagram]

Figure 2. List with an anchor.

The class diagram in Figure 2 shows a list composed of an anchor of class `List` and a number of elements instantiating class `Element`. The set of elements contained in a list is denoted by `elements`; it is defined with the help of auxiliary function `succOf` collecting in set `Acc` all successors of element `el`:

```plaintext
context List def :
    self.elements : Collection(Element) = if self.first = Set{} then Set{} else self.first.succOf(Set{self.first}) endif
context Element def :
    self.succOf(Acc : Collection(Element)) : Collection(Element) = 
        if self.next = Set{} or Acc->includes(self.next) then Acc
        else self.next.succOf(Acc->union(Set{self.next})) endif
```

Here we consider only finite acyclic lists. This constraint is expressed by an invariant saying that a nonempty list must contain an element that does not have a successor.

```plaintext
context List inv no_loops :
    self.elements->notEmpty() implies self.elements->exists(el | el.next = Set{})
```

We may also require that different lists are disjoint, i.e., two lists corresponding to two different anchor objects do not share elements:

```plaintext
context self1, self2 : List inv no_sharing :
    self1 <> self2 implies self1.elements->excludesAll(self2.elements)
```

The next invariant says that attribute `total` is equal to the sum of the values of attribute `x` for the corresponding list elements.

```plaintext
context List inv inv_total : self.total = self.elements.x->sum()
```

The method `insert` inserts an element `e` into a list. We use `elements@pre` to denote the set of all list elements which exist in the pre-state.

```plaintext
context List::insert(e : Element)
    post : self.elements = self.elements@pre->including(e)
    modifies : self.total, self::first, self.elements::next
```
The modifies-clause restricts changes to attribute total of the actual implicit parameter, to the actual list’s anchor and to association-end next of the list’s elements. This clause in conjunction with the first part of the post-condition implies that the element e is added, but no elements can be removed (cf. [26]).

The following constraint specifies method award which awards the amount s to a list element.

```plaintext
context Element::award(s : Real)
post : self.x = self.x@pre + s
modifies mod_award : self::x
```

If insert satisfies only its post-condition, then it may change the value of total in the actual list in an inappropriate way or violate the invariant inv_total in some other lists. It may also violate the invariants no_loops and no_sharing. Similarly the method award may violate the invariant inv_total.

![Figure 3. Concrete lists.](image)

Figure 3 shows an object diagram corresponding to the class diagram in Figure 2. It contains two lists. In this case, the term l2.elements has the value \{leA, leB\} and the term le3.succOf({le2}) has the value \{le2, le3, le4\}. This is due to the fact that the resulting set contains the initial accumulator and with every successor the accumulator is extended by adding the successor. Both lists share element le4. Thus, awarding a certain amount of money or adding a new element to le1 may potentially violate invariant inv_total in case of le2. If the element le4 is present in the list l2 (as shown in Figure 3), then invariant no_sharing is violated.

4. Extracting Languages from OCL-Terms

In this section we show how to extract a context free language from an OCL-term. Words of such a language define paths which can be navigated during evaluation of the corresponding term. In the
first subsection we present a restricted subset of OCL and its simplified type system. In the second subsection we show how to extract partial words from OCL-terms. In the third subsection we show that term decomposition corresponds to word decomposition. In the fourth subsection we associate context free grammars with terms. We use examples introduced in the previous section to illustrate these concepts.

4.1. Restricted OCL-Syntax and its Typesystem

In this subsection we restrict OCL to its core features. We show how to eliminate class attributes from recursive definitions. We define a syntax of the restricted version of OCL using an EBNF-grammar and present a simplified type system which abridges the type hierarchy of OCL.

In OCL, object attributes, class-attributes and association-ends are called ‘properties’ (cf. [38]). We call object attributes and association-ends ‘object-properties’ and class attributes ‘class-properties’. We treat class attributes in a separate way. For the sake of simplicity, we assume that object-properties have only one argument and that this argument must be of object type. However, in general, an object-property can have multiple arguments (see the UML standard [38]).

OCL-terms may contain recursively defined function symbols. Their OCL-definition may have the form:

\[
\text{context } C_i \\
\text{def : } f_i(y_1, \ldots, y_k) = F_i(y_1, \ldots, y_k) \\
\ldots \\
\text{context } C_m \\
\text{def : } f_m(y_1, \ldots, y_k) = F_m(y_1, \ldots, y_k)
\]

where \( C_i \) are class names and \( F_i \) are OCL-terms containing attributes, association-ends and the recursively defined function symbols \( f_1, \ldots, f_m \) (see for example subsection 3.3). We assume that the formulas on the right hand side do not include class attributes (for example \( t_x \), see subsection 3.2). If definition \( F(y_1, \ldots, y_k) \) of symbol \( f(y_1, \ldots, y_k) \) includes a class attribute \( C.c \), then we can transform \( f \) into \( g(z, y_1, \ldots, y_k) \), where \( z \) is a fresh variable, and \( F \) into \( G(z, y_1, \ldots, y_k) \), where \( G \) is obtained from \( F \) by replacing all occurrences of class attribute \( C.c \) by \( z \). In this way after eliminating all class attributes, we obtain a more general definition. However, the function symbol \( f \) can be defined by \( G[C.c/z] \); i.e., it can be obtained from term \( G \) by substituting for variable \( z \) constant \( C.c \) formalizing the class attribute.

Dealing with all OCL features is not feasible in a paper like this. Therefore we consider a restricted form of OCL-terms containing object-properties, recursively defined function symbols and core operators. The following grammar defines variables \(<\text{var}>\), untyped terms \(<\text{uT}>\), types \(<\text{type}>\) and typed terms \(<\text{tT}>\). The nonterminal \(<\text{a}>\) corresponds to object-properties, such as for example name and pilots of class Airline (see Fig.1), and \(<\text{f}>\) corresponds to recursively defined function symbols. The nonterminal \(<\text{class}>\) corresponds to class names, e.g., Airline.

\[
<\text{var}> ::= x_1 | x_2 | \ldots \\
<\text{uT}> ::= <\text{var}> | 0 | 1 | \text{true} | \text{false} | <\text{uT}> - <\text{uT}> | <\text{uT}> . <\text{a}> | f([<\text{uT}>].<\text{uT}>]) | \\
\text{if } <\text{uT}> \text{ then } <\text{uT}> \text{ else } <\text{uT}> \text{ endif } | <\text{uT}> . \text{sum}() | \text{Set}([<\text{uT}>].<\text{uT}>]) | \\
<\text{uT}> . \text{collect}(<\text{var}>[<\text{type}>]| <\text{uT}>) | <\text{uT}> . \text{union}(<\text{uT}>) | <\text{uT}> = <\text{uT}> | \\
\[ <uT> \rightarrow \text{includes}(<uT>) \mid <uT> < <uT> \]

\[ \text{stype} ::= \text{Boolean} \mid \text{Integer} \mid \text{Real} \mid \text{class} \]

\[ \text{type} ::= \text{Collection}(\text{stype}) \mid \text{OCLAny}, \]

\[ <T> ::= <uT> : <type> \]

Types Boolean, Integer and Real are called ‘basic OCL types’. We call these types and the corresponding collections ‘BOT’. We do not distinguish between different types of collection such as Set, Bag and Sequence (cf. [38]). The type OCLAny encompasses all other types. We assume that
\[ \text{Collection}(	ext{Collection}(T)) = \text{Collection}(T) \]
It should be noted that several OCL-operators can be defined using above listed ones, for example +, not, implies and or, if sequential evaluation is assumed; Kuhlmann and Gogolla [28] discuss this issue more thoroughly.

Typing is always done in relation to a class model, which defines a hierarchy of classes with object properties (see for example Figure 1). We assume that terms of the form \(<\text{var}> : <\text{type}>\) are well typed and define the following typing rules for complex terms:

\[ t : \text{Collection}(A), a \text{ is an object-property of class } A \text{ with values of type } \text{Collection}(B) \]

\[ t.a : \text{Collection}(B) \]

\[ v : \text{Boolean}, t_1 : C, t_2 : C \]

\[ f : T_1 \times \cdots \times T_n \to T, t_1 : T_1, \ldots, t_n : T_n \]

\[ \text{if } v \text{ then } t_1 \text{ else } t_2 \text{ endif : } C \]

\[ f(t_1, \ldots, t_n) : T \]

\[ t : \text{Collection}(A) \text{ and } t : \text{OCLAny} \]

\[ t : A \]

\[ B \text{ is a subclass of } A, t : B \]

\[ t_1 : \text{Collection}(A), t_2 : \text{Collection}(C) \]

\[ t_1 \to \text{union}(t_2) : \text{Collection}(C) \]

\[ \text{Set}\{t_1, \ldots, t_n\} : \text{Collection}(C) \]

\[ t_1 \to \text{collect}(y : B \mid t_2) : \text{Collection}(C) \]

We skip obvious typing rules for the following operators: =, <, includes, 0, 1, −, true, false. We assume that if term \(t\) is of type \(\text{Collection(Real)}\), then \(t \to \text{sum}()\) is well typed, and similarly for type Integer. We say that term \(t\) is well typed and has type \(T\) if it can be proved using the above mentioned assumptions and typing rules that \(t : T\).

Below for simplicity, we assume that except if then else endif all those operators are interpreted as strict functions. Under the assumption that \(\text{forall} \) is a strict operator, we can define it using \(\text{collect}\), i.e., a term of the form \(t_1 \to \text{forall}(x \mid t_2)\) can be interpreted by \(\text{not}(t_1 \to \text{collect}(x \mid t_2) \to \text{includes}(\text{false}))\). The second term looks a bit strange, but it makes sense in OCL. Term \(t_1\) results in a set of objects, or values in general. Term \(t_2\) must be of the boolean type. \(\text{forall}\) evaluates term \(t_2\) for all objects in the set defined by \(t_1\). It returns \(\text{true}\) if the value of \(t_2(x)\) is true for every object in that set. If for an object the value of \(t_2\) is \(\text{false}\), then \(\text{forall}\) returns \(\text{false}\). Operation \(\text{collect}\) collects values of \(t_2\) for the elements of \(t_1\). If \(t_2\) returns \(\text{false}\) for one of the objects defined by \(t_1\), then the resulting set of values includes \(\text{false}\). If \(t_1\) is undefined, or \(t_2\) is undefined for one of the objects, then terms \(t_1 \to \text{forall}(x \mid t_2(x))\) and \(t_1 \to \text{collect}(x \mid t_2)\) are undefined as well. Similarly we have the following equivalences:

\[ t_1 \to \text{includesAll}(t_2) \text{ and } t_2 \to \text{forall}(o \mid t_1 \to \text{includes}(o)), \]
t₁->excludesAll(t₂) and t₂->forAll(o | not(t₁->includes(o))).

Variable y in term t₁->collect(y | t₂) is bound. Similarly, y is bound in t₁->forAll(y | t₂). The concept of bound variables comes from first-order logic. Quantified variables are called ‘bound’ and unbound variables are called ‘free’. Without loss of generality, we assume that for every term the set of bound variables and the set of free variables are disjoint. Moreover, we assume that variables bound by different collect operations are different. These assumptions can be made without loss of generality since bound variables can be renamed using the so-called alpha-conversion.

As mentioned in the previous section, OCL-constraints are always defined in a context of a class from a class model. If a term t(self,...) is defined in the context of class C, then it can be equivalently defined by typing the implicit parameter: t(self : C,...). As it is common in logic, free variables in formulas are considered to be all-quantified. Thus an invariant I defined in the context of class C can be expressed in the form C.allInstances()->forAll(self : C | I), where allInstances() is a property returning all instances of a given class (cf. [38]).

4.2. Extraction of Partial Words from OCL-Terms

In this subsection, we show how to extract sets of partial words from OCL-terms. In fact we need two kinds of such sets. One of them allows to navigate to term values. The other is prefix-closed and allows to determine those values. On their basis we construct context free languages.

For a class model CM, we define the set of terminal symbols corresponding to its object-properties, i.e., \( T = \{ a | a \text{ is an object property in } CM \} \). In case of Figure 2, \( T \) has the form \{total, first, next, x\}. For every inductively defined function symbol \( f_1 \) and its variable \( y_j \), we introduce two nonterminal symbols \( f_{1,y_j} \) and \( f_{1,y_j}^{C} \), e.g., for elements and the implicit parameter self we introduce \( \text{elements}_{\text{self}} \) and \( \text{elements}_{\text{self}}^{C} \). For every object-property \( a \), we introduce terminal symbol \( a \) and nonterminal symbol \( a^{C} \). E.g., for next we introduce \( \text{next} \) and \( \text{next}^{C} \). We write \( f \) for an arbitrary recursively defined function symbol and \( F \) for the corresponding definition. We define also the set \( N_{t_0} \) containing all nonterminals of the form \( a^{C} \), nonterminals of the form \( f_{1,y_j} \) and \( f_{1,y_j}^{C} \) where \( y_j \) is a variable of \( f_1 \), e.g., \( N_{t_0} = \{ \text{total}^{C}, \ldots, x^{C}, \text{elements}_{\text{self}}, \text{elements}_{\text{self}}^{C}, \ldots \} \).

A partial word is a sequence of terminals and nonterminals. A total word is a sequence of terminal symbols. A set of partial words is called partial language and a set of words is called language. Note that we use the same character to denote a property and the corresponding terminal (function symbol and the corresponding nonterminal, resp.), but we differentiate between them using different fonts. In general, we use serif font to denote terminals and italic to denote their formalizations.

For an OCL-term \( t \) of the restricted syntax and a free variable \( x \) we define two sets of partial words. The first set \( P_{re}L_{x}^{O}(t) \) defines paths navigated during term evaluation which start at an actual parameter corresponding to \( x \) and lead to objects determining the value of \( t \). The second set \( P_{re}L_{x}^{C}(t) \) defines all paths relevant for determining the value of \( t \) which start at an actual parameter corresponding to \( x \). We call a set of the form \( P_{re}L_{x}^{O}(t) \) open pre-language. We call a set of the form \( P_{re}L_{x}^{C}(t) \) closed pre-language since it defines all paths navigated when a term is evaluated; this includes evaluation of subterms and in particular boolean conditions in if-statements.
Definition 1. Let \( t \) be an OCL-term of the restricted syntax as described above and let \( x \) be a variable.

1. Let \( t \) be a variable. If \( x \) is identical with \( t \), then \( \mathcal{P}_{x}^{O}(t) = \text{def} \{ e \} \) and \( \mathcal{P}_{x}^{C}(t) = \text{def} \{ \} \); in the other case \( \mathcal{P}_{x}^{O}(t) = \text{def} \emptyset \) and \( \mathcal{P}_{x}^{C}(t) = \text{def} \emptyset \).

2. Let \( t \) have the form \( u \cdot b \), for some term \( u \) and an object-property \( b \). Then \( \mathcal{P}_{x}^{O}(t) = \text{def} \mathcal{P}_{x}^{O}(u) \cdot b \) and \( \mathcal{P}_{x}^{C}(t) = \text{def} \mathcal{P}_{x}^{O}(u) \cup \mathcal{P}_{x}^{C}(u) \).

3. Let \( t \) have the form \( f(t_{1}/y_{1}, \ldots, t_{k}/y_{k}) \), for a recursively defined function \( f \) and terms \( t_{1}, \ldots, t_{k} \). The open pre-language is defined as follows:
   \[
   \mathcal{P}_{x}^{O}(t) = \text{def} \mathcal{P}_{x}^{O}(t_{1})f_{y_{1}} \cup \ldots \cup \mathcal{P}_{x}^{O}(t_{k})f_{y_{k}}. 
   \]
   The closed pre-language is defined as follows:
   \[
   \mathcal{P}_{x}^{C}(t) = \text{def} \mathcal{P}_{x}^{O}(t) \cup \mathcal{P}_{x}^{C}(t_{1}) \cup \ldots \cup \mathcal{P}_{x}^{C}(t_{k}) \).

4. If \( t \) has the form \( \text{if } v \text{ then } t_{1} \text{ else } t_{2} \text{ end if} \), then
   \[
   \mathcal{P}_{x}^{O}(t) = \text{def} \mathcal{P}_{x}^{O}(t_{1}) \cup \mathcal{P}_{x}^{O}(t_{2}) \text{ and } \mathcal{P}_{x}^{C}(t) = \text{def} \mathcal{P}_{x}^{O}(v) \cup \mathcal{P}_{x}^{C}(t_{1}) \cup \mathcal{P}_{x}^{C}(t_{2}) \).

5. Let \( t \) have the form \( t_{1}\triangleright\text{collect}(y \mid t_{2}) \) and let \( x \) be a free variable occurring in \( t \), then
   \[
   \mathcal{P}_{x}^{O}(t) = \text{def} \mathcal{P}_{x}^{O}(t_{1}) \cup \mathcal{P}_{x}^{O}(t_{2}) \text{ and } \mathcal{P}_{x}^{C}(t) = \text{def} \mathcal{P}_{x}^{O}(t_{1}) \cup \mathcal{P}_{x}^{C}(t_{2}) \).

6. If \( t \) has the form \( t_{1} \square t_{2} \), for \( \square \) being a symbol of the form and, =, <, or the form \( t_{1} \rightarrow \text{union}(t_{2}) \) or \( t_{1} \rightarrow \text{includes}(t_{2}) \), then
   \[
   \mathcal{P}_{x}^{O}(t) = \text{def} \mathcal{P}_{x}^{O}(t_{1}) \cup \mathcal{P}_{x}^{O}(t_{2}) \text{ and } \mathcal{P}_{x}^{C}(t) = \text{def} \mathcal{P}_{x}^{C}(t_{1}) \cup \mathcal{P}_{x}^{C}(t_{2}) \).

7. If \( t \) has the form \( \text{not}(x) \) or \( x \rightarrow \text{sum}() \), then \( \mathcal{P}_{x}^{O}(t) = \text{def} \mathcal{P}_{x}^{O}(x) \) and \( \mathcal{P}_{x}^{C}(t) = \text{def} \mathcal{P}_{x}^{C}(x) \). Similarly, if \( t \) has the form \( \text{Set}\{r_{1}, \ldots, r_{n}\} \), then \( \mathcal{P}_{x}^{O}(t) = \text{def} \mathcal{P}_{x}^{O}(r_{1}) \cup \ldots \cup \mathcal{P}_{x}^{O}(r_{n}) \) and \( \mathcal{P}_{x}^{C}(t) = \text{def} \mathcal{P}_{x}^{C}(r_{1}) \cup \ldots \cup \mathcal{P}_{x}^{C}(r_{n}) \).

8. If \( t \) has the form \( \text{Set}\{\} \), or \( 0 \), or \( 1 \), or \( \text{true} \), then \( \mathcal{P}_{x}^{O}(t) = \text{def} \emptyset \) and \( \mathcal{P}_{x}^{C}(t) = \text{def} \emptyset \).

The value returned by operations mentioned in items (4) – (8) does not depend on the current state; therefore we call them state independent. As mentioned above, we regard the operations treated in items (5) – (8) as strict. The closed pre-language \( \mathcal{P}_{x}^{C}(t) \) contains prefixes of the open-pre language \( \mathcal{P}_{x}^{O}(t) \); but it may be more extensive, due to (4) and (5). Note that that the language generated for terms obtained by binary operations such as \( \neg \) (see point 6) are defined to be unions of the languages corresponding to their arguments.

Below we write \( t \) for a tuple of terms of the form \( (t_{1}, \ldots, t_{k}) \). Similarly we write \( u[t/x] \) for the substitution \( u[t_{1}/y_{1}, \ldots, t_{k}/y_{k}] \). If the set of free variables of term \( u \) is included in set \( \{x_{1}, \ldots, x_{k}\} \), then we write \( u(x_{1}, \ldots, x_{k}) \); in this case we denote the corresponding substitution using round brackets, i.e., \( u(t/y) \).

---

\footnote{Recall that the sets of free variables and of bound variables are assumed to be disjoint; in particular \( x \) is different from \( y \).}
We show how to extract partial words from the definition of \texttt{elements} in the list example (see subsection 3.3). Basically, those words correspond to paths navigated through when the term is evaluated. The definition of \texttt{elements} includes function \texttt{succ0f}. This function has two arguments, therefore there are two nonterminals corresponding to its arguments. Let \( t \) be the body of the definition of \texttt{elements}. The corresponding closed pre-language has the form:

\[
P_{\text{reL}}(t) = \{ \epsilon, \text{first}, \text{first succOf}, \text{first succOf} \}\]

We use variables \( \text{self} \) and \( \text{Acc} \) to refer to the arguments of \texttt{succ0f} occurring in the recursive definition of this function. In case of \( P_{\text{reL}}(t) \), the first part corresponding to the boolean condition is eliminated, i.e., \( P_{\text{reL}}(t) = \{ \text{first succOf}, \text{first succOf} \text{Acc} \} \) (see the definition of \( P_{\text{reL}}(t) \)).

Let \( t \) be the body of the definition of \texttt{succ0f}. The closed pre-language \( P_{\text{reL}}(t) \) corresponding to variable \( \text{self} \) has the form:

\[
P_{\text{reL}}(\text{self}.next = \text{Set}) \cup P_{\text{reL}}(\text{Acc} -> \text{includes}(\text{self}.next)) \cup P_{\text{reL}}(\text{Acc} \cup \text{self}.next.succOf(\text{Acc} -> \text{union}(\text{Set} \text{self}.next)))
\]

\[
= \{ \epsilon, \text{next succOf}, \text{next succOf} \text{Acc} \}.
\]

Note that set \( P_{\text{reL}}(\text{Acc}) \) is empty since the variables \( \text{self} \) and \( \text{Acc} \) are different. The closed pre-language corresponding to the second argument has the form \( P_{\text{reL}}(\text{Acc}) = \{ \epsilon, \text{succOf} \text{Acc} \} \).

4.3. Term Structure versus Word Decomposition

In this subsection we show that term decomposition is closely related to decomposition of the corresponding words. We prove a number of properties which are then used to characterize context free languages corresponding to OCL-terms.

The following lemma lists properties of open and closed pre-languages. The first two points follow directly from the definition of pre-languages. Point 3 says that a word belongs to a closed pre-language of a term if, and only if, the term has a minimal subterm \( s \) such that the word belongs to the open pre-language of \( s \) or \( s \) is of the form \texttt{collect} and the word can be decomposed; the second case is shown in Figure 4. The free variable, where the word starts, is indicated by a continuous rectangle and the bound variable is indicated by a dotted rectangle. The word is indicated by a line; its dotted part indicates continuation. Note that the same variable can occur multiple times in a term. The previous point and (4) relate closed and open pre languages. It should be noted that in (3) – (6) we substitute only for free variables. We use the following lemma to prove various properties of closed pre-languages.

Lemma 4.1. Let \( t \) be an OCL-term of the restricted syntax and \( x \) its free variable.

1. \( P_{\text{reL}}(t) \subseteq P_{\text{reL}}(t) \)

2. If \( s \) is a subterm of \( t \), then \( P_{\text{reL}}(s) \subseteq P_{\text{reL}}(t) \)
3. Let \( w \) be a partial word. \( w \in \text{Pre} L^C_x(t) \) if, and only if, there exists a subterm \( s \) of \( t \) such that either \( w \in \text{Pre} L^O_x(s) \), or \( s \) has the form \( t_1 \rightarrow \text{collect}(y \mid t_2) \), \( w = w_1w_2, w_1 \in \text{Pre} L^O_x(t_1), w_2 \in \text{Pre} L^C_x(t_2) \) and \( |w_2| > 0 \)

4. Let \( w \) be a partial word, let \( w \in \text{Pre} L^C_x(t) \), let \( t \) have the form \( t_1 \rightarrow \text{collect}(x_2 \mid t_2) \), let \( w = w_1v, w_1 \in \text{Pre} L^O_x(t_1), v \in \text{Pre} L^C_x(t_2) \) and \( |v| > 0 \). Then there exist terms \( u_2, t_3, u_3, \ldots, t_n, u_n \) such that:
   - \( t_1 \) has a subterm of the form \( u_1 \rightarrow \text{collect}(x_{i+1} \mid t_{i+1}) \), for \( i = 2, \ldots, n-1 \), and \( u_n \) is a subterm of \( t_n \)
   - \( v \) can be presented in the form \( w_2 \ldots w_n \)
   - \( w_i \in \text{Pre} L^O_x(u_i), for i = 2, \ldots, n, and |w_n| > 0 \)

5. Let \( x_1, \ldots, x_k \) be free variables of \( t \). A word \( w \) belongs to \( \text{Pre} L^O_x(t(t_{i_1}/x_1, \ldots, t_k/x_k)) \) if, and only if, \( w = w_1w_2 \), for some \( i, w_1 \in \text{Pre} L^O_x(t_{i_1}) \) and \( w_2 \in \text{Pre} L^O_x(t) \)

6. Let \( r[t/z] \) be an OCL-term and let \( z \) be a free variable of \( r \). If \( w_1 \in \text{Pre} L^O_x(t) \) and \( w_2 \in \text{Pre} L^C_x(r[t/z]) \), then \( w_1w_2 \in \text{Pre} L^C_x(r[t/z]) \)

7. Let \( A_1 \ldots A_{n-1}A_n \) be a sequence of terminals and nonterminals. If the partial word \( A_1 \ldots A_{n-1}A_n \) belongs to \( \text{Pre} L^C_x(t) \), then the partial words \( A_1 \ldots A_{i-1}A_i \) belong to \( \text{Pre} L^C_x(t) \), for every \( i < n \)

**Proof:**
Conditions (1) and (2) follow by structural induction from the definition of \( \text{Pre} L^O_x(t) \) and \( \text{Pre} L^C_x(t) \).

In case of (3), the implication from right to left follows directly from (1) and (2). We prove by structural induction on the complexity of \( t \) that the left hand side of the equivalence implies its right hand side. This implication clearly holds for variables and constants (cf. items (1) and (8) of definition (1)). We call such terms ‘atomic’. Let \( t \) be a non-atomic term, let the property hold for all proper subterms of \( t \) and let \( w \in \text{Pre} L^C_x(t) \). If a proper subterm \( s \) of \( t \) exists and \( w \in \text{Pre} L^C_x(s) \), then (3) follows from the inductive assumption and point (2). Let \( t \) do not have proper subterms \( s \) such that \( w \in \text{Pre} L^C_x(s) \); we call it here the ‘minimality assumption’. If \( t \) has the form \( t_1 \rightarrow \text{collect}(y\)
The proof of (4) follows by structural induction on the number of collect operations in the term \( t \). If there is only one such an operation, then (4) follows directly from (3) and from the definition of \( \text{Pre}L_x^C(t_1 \rightarrow \text{collect}(z \mid t_2)) \). Let it holds for all terms having less than \( m \) occurrences of collect, let \( t \) contain \( m \) occurrences and let the assumption of (4) be satisfied. If \( v \in \text{Pre}L_{x_2}^O(u_2) \), for a subterm \( u_2 \) of \( t_2 \), then the property clearly holds. If not, then from (3) it follows that \( t_2 \) has a subterm of the form \( u_2 \rightarrow \text{collect}(x_3 \mid t_3) \) such that \( v \) can be presented in the form \( w_2v' \), \( w_2 \in \text{Pre}L_{x_2}^O(u_2), v' \in \text{Pre}L_{x_3}^C(t_3) \) and \( |v'| > 0 \). In this case the property follows from the inductive assumption. Figure 5 shows this case up to term \( t_3 \).

Point (5) is also proved by structural induction. It should be noted that no free variable becomes bound due to this substitution since by assumption the sets of free and bound variables are disjoint. If \( t \) is a variable, then \( \text{Pre}L_x^O(t) = \{ e \} \), and \( w \in \text{Pre}L_x^O(t(t/x)) \) if, and only if, there is an \( i \), such that \( x_i \) is identical with \( t \) and \( w \in \text{Pre}L_x^O(t_1) \). If \( t \) is a constant, then the corresponding language is empty due to the definition of the pre-language. Let (5) hold for all proper subterms of \( t \). If \( t \) has the form \( f(s_1/y_1, \ldots, s_k/y_k) \), for a recursively defined function \( f \), then \( w \in \text{Pre}L_x^O(t(t/x)) \) if, and only if, there exists a \( j \), such that \( w \in \text{Pre}L_x^O(s_j(t/x))f_{y_j} \). From the inductive assumption and the definition of the pre-language follows that \( w \in \text{Pre}L_x^O(t(t/x)) \) if, and only if, there exist \( i \) and \( j \) such that \( w \) can be decomposed into \( w_1w_2f_{y_j} \), where \( w_1 \in \text{Pre}L_x^O(t_1), w_2 \in \text{Pre}L_x^O(s_j) \). In case of \( t \) having the form \( s \rightarrow \text{collect}(y \mid s) \), then \( w \in \text{Pre}L_x^O(t(t/x)) \) if, and only if, \( w \in \text{Pre}L_x^O(s_1(t/x)) \) or \( w \) can be decomposed into \( v_1v_2 \), where \( v_1 \in \text{Pre}L_x^O(s_1(t/x)) \) and \( v_2 \in \text{Pre}L_y^O(s_2) \). In the first case, the proof follows from the inductive assumption. In the second case, the inductive assumption implies that \( v_1 \in \text{Pre}L_x^O(s_1(t/x)) \) if, and only if, there exist an \( i \) such that \( v_1 \) can be decomposed into \( v_1v_2v_3 \) where \( v_1 \in \text{Pre}L_x^O(t_1) \) and \( v_1 \in \text{Pre}L_x^O(s_1) \). Thus, we can define \( w_1 = v_1v_2 \) and \( w_2 = v_1v_2v_3 \). It is essential that variables \( x_i \) are
free and that we substitute only for free variables. Figure 6 shows this case; $y$ is the bound variable. For other state independent operations, the inductive proof follows directly from the definition of the pre-language.

![Figure 6. A word within a complex collect-term.](image)

To prove (6), we assume that the left hand side of the implication is satisfied. If $w_2 \in \mathcal{P}_{\text{re}} \mathcal{L}_x^C(t)$, then (3) implies that there is a subterm $s$ of $t$ such that either $w_2 \in \mathcal{P}_{\text{re}} \mathcal{L}_x^O(s)$, or $s$ has the form $s_1 \to \text{collect}(y | s_2)$, $w_2 = w_{21}w_{22}$, $w_{21} \in \mathcal{P}_{\text{re}} \mathcal{L}_x^O(s_1)$ and $w_{22} \in \mathcal{P}_{\text{re}} \mathcal{L}_x^C(s_2)$. In the first case, the property follows from (5). In the second case, from (5) it follows that $w_{21}w_{22} \in \mathcal{P}_{\text{re}} \mathcal{L}_x^O(s_1[t/z])$. The definition of the closed pre-language implies that $w_1w_{21}w_{22} \in \mathcal{P}_{\text{re}} \mathcal{L}_x^C(s_1[t/z] \to \text{collect}(y | s_2))$. Note that $z$ is a free variable and $y$ is bound.

We prove (7) by structural induction for $i = n - 1$. It clearly holds for atomic terms. Let the property hold for all proper subterms of $t$ and let $A_1 \ldots A_{n-1}A_n$ belong to $\mathcal{P}_{\text{re}} \mathcal{L}_x^C(t)$. From (3) it follows that there is a subterm $s$ of $t$ such that $A_1 \ldots A_n$ belongs to $\mathcal{P}_{\text{re}} \mathcal{L}_x^O(s)$ or $s$ is of the form $s_1 \to \text{collect}(y | s_2)$, $A_1 \ldots A_{j-1} \in \mathcal{P}_{\text{re}} \mathcal{L}_x^O(s_1)$ and $A_j \ldots A_n \in \mathcal{P}_{\text{re}} \mathcal{L}_x^C(s_2)$, for a $j \leq n$. Let $s$ be a minimal subterm with this property. In the collect-case, $A_j \ldots A_n$ is different from $\epsilon$. From the inductive assumption it follows that $A_j \ldots A_{n-1}$ belongs to $\mathcal{P}_{\text{re}} \mathcal{L}_x^C(s_2)$ and consequently $A_1 \ldots A_{n-1} \in \mathcal{P}_{\text{re}} \mathcal{L}_x^C(s)$. If the partial word belongs to the open pre-language $\mathcal{P}_{\text{re}} \mathcal{L}_x^O(s)$, then there are three possible cases: the collect case treated above, or $s$ is of the form $\tau$, a, for a property a, or $s$ is of the form $f(t_{j/y_1}, \ldots, t_k/y_k)$, for a recursively defined function $f$. In case of $\tau$, we can consider the term $\tau'$; from the definition of the pre-language it follows that $A_1 \ldots A_{j-1} \in \mathcal{P}_{\text{re}} \mathcal{L}_x^O(\tau) \subseteq \mathcal{P}_{\text{re}} \mathcal{L}_x^C(\tau) \subseteq \mathcal{P}_{\text{re}} \mathcal{L}_x^C(t)$ due to (1) and (2). The second case is analogous. We can repeat this procedure until the partial word is short enough, i.e., we can prove this property for every $i < n$. □

### 4.4. Context Free Grammar of OCL-Term

In this subsection we show how to associate context free languages with OCL-terms. Those languages define paths traversed when evaluating the corresponding terms. The extracted languages are context free since in first-order logic and similar logical systems substitution is allowed only for free variables and there is no $\beta$-conversion.

For an arbitrary partial language $L$, the set of the corresponding prefixes is defined as follows: $\mathcal{L}_{\text{Init}}(L) =_{\text{def}} \{ w | \exists w_1, w w_1 \in L \}$. $L$ is prefix-closed if $\mathcal{L}_{\text{Init}}(L) = L$. The division of $L$ by a terminal is
defined as follows: \( L/a =_{\text{def}} \{ w \mid wa \in L \} \). It is well known that if \( L \) is a context free language, then \( T_{\text{init}}(L) \) and \( L/a \) are context free as well (cf. e.g. [23]).

A grammar has the form \( G = (T, Nt, s, \rightarrow) \), where \( T \) is a set of terminal symbols, \( Nt \) is a set of nonterminal symbols, \( s \) is a start symbol and \( \rightarrow \) is a reduction relation. We say that word \( w' \) can be derived from \( w \) and write \( w \Rightarrow w' \) if \( w' \) can be obtained from \( w \) by application of \( \rightarrow \) to a nonterminal symbol occurring in \( w \). \( \Rightarrow^n \) denotes \( n \)-step reduction, \( \Rightarrow^+ \) denotes the transitive closure of \( \Rightarrow \), and \( \Rightarrow^* \) denotes the reflexive and transitive closure of \( \Rightarrow \). For grammar \( G \), we define the corresponding language \( L(G) =_{\text{def}} \{ w \mid s \Rightarrow^* w \} \cap T^* \), where the set \( T^* \) contains sequences of terminal symbols. We use capital letter \( A \) to denote arbitrary terminals and nonterminals. Below, we distinguish between different reduction relations by using subscripts. However, we skip those subscripts if it is clear which reduction relation is meant.

For the empty word \( \epsilon \) and a letter \( A \) having the form the \( A \) or \( f \), we denote by \( A^C \) a new nonterminal obtained from \( A \) by applying the superscript \( C \); i.e., if \( A \) has the form a, \( \epsilon \) or \( f \), then \( A^C \) denotes \( a^C \), \( \epsilon^C \) or \( f^C \), respectively. We assume that \( A^{CC} \) is equal to \( A^C \). For a word \( w = A_1 \ldots A_n \) such that \( n > 0 \), by \( w^C \) we denote \( A_1 \ldots A_{n-1}^C \). Similarly for a partial language \( L \), \( L^C \) denotes \( \{ w^C \mid w \in L \} \).

For the inductively defined function symbols \( f_i \) and the corresponding nonterminals (see subsection 4.2), we define reduction relation \( \rightarrow_C \) including the following reduction rules:

\[
\begin{align*}
&f_{i_1yj} \rightarrow_C P_{\text{re}}L_{yj}^C(F_1) \quad \text{and} \quad f_{i_2yj}^C \rightarrow_C P_{\text{re}}L_{yj}^C(F_1)^C
\end{align*}
\]

where \( y_j \) is a free variable occurring in the definition of \( f_i \). We say that term \( u \) is obtained by one-step unwinding of term \( t \) if \( u \) is obtained by applying a recursive definition to the corresponding function symbol occurring in \( t \); i.e., by replacing a symbol \( f_i \) by the corresponding definition \( F_i \). We say that \( u \) is obtained by unwinding of term \( t \) if \( u \) is obtained by a number of one-step unwinderings. For example, the nonterminal \( \text{elements} \) can be replaced by its definition and then in the resulting term \( \text{succOf} \) can be replaced by the definition of \( \text{succOf} \).

To illustrate how the grammar is defined, we consider the list example (see subsections 3.3 and the end of subsection 4.2). We show how to extract reduction rules from the definition of \( \text{elements} \).

The reduction rules have the form:

\[
\begin{align*}
\text{elements}_\text{self} & \rightarrow_C \epsilon \mid \text{first} \mid \text{first succOf}_\text{self} \mid \text{first succOf}_\text{Acc} \\
\text{elements}_\text{self}^C & \rightarrow_C \epsilon \mid \text{first} \mid \text{first succOf}_\text{self}^C \mid \text{first succOf}_\text{Acc}^C
\end{align*}
\]

The rules for terminals decorated with the superscript \( C \) are as follows:

\[
\begin{align*}
\text{elements}_\text{self} & \rightarrow_C \epsilon \mid \text{first} \mid \text{first succOf}_\text{self} \mid \text{first succOf}_\text{Acc} \\
\text{elements}_\text{self}^C & \rightarrow_C \epsilon \mid \text{first} \mid \text{first succOf}_\text{self}^C \mid \text{first succOf}_\text{Acc}^C
\end{align*}
\]

The reduction rules corresponding to nonterminals \( \text{succOf}_\text{self} \) and \( \text{succOf}_\text{Acc} \) have the form:

\[
\begin{align*}
\text{succOf}_\text{self} & \rightarrow_C \epsilon \mid \text{next} \mid \text{next succOf}_\text{self} \mid \text{next succOf}_\text{Acc} \quad \text{and} \quad \text{succOf}_\text{Acc} \rightarrow_C \epsilon \mid \text{succOf}_\text{Acc}
\end{align*}
\]

The last rule is redundant.

The following lemma relates application of recursive definitions to word derivation. It says that the unwinding of a term corresponds to derivation of words from the corresponding partial language. It shows that for an OCL-term, the unwinding of recursively defined symbols occurring in the term is matched by the reduction of the corresponding words and vice versa.

**Lemma 4.2. (Reduction Lemma)**

Let \( t \) be an arbitrary OCL-term of the restricted syntax as defined in subsection 4.2, let \( x \) be a free variable of \( t \) and let \( w' \) be an arbitrary partial word. Then the following properties hold:
1. There exists a partial word \( w \in \mathcal{P} \mathcal{L}_x^O(t) \) such that \( w \Rightarrow^* \mathcal{C} w' \) if, and only if, there exists an unwinding \( u \) of \( t \) such that \( w' \in \mathcal{P} \mathcal{L}_x^O(u) \).

2. There exists a partial word \( w \in \mathcal{P} \mathcal{L}_x^C(t) \mathcal{C} \) such that \( w \Rightarrow^* \mathcal{C} w' \) if, and only if, there exists an unwinding \( u \) of \( t \) such that \( w' \in \mathcal{P} \mathcal{L}_x^C(u) \mathcal{C} \).

**Proof:**

We prove (1) first. It is enough to consider the case of one-step unwindings and reductions. Indeed, let (1) hold for one-step reductions. If the implication \( \Rightarrow \) holds for \( n \) reduction steps and if \( w \Rightarrow_{\mathcal{C}} \mathcal{C} w' \), then there is an unwinding \( u \) of \( t \) such that \( w' \in \mathcal{P} \mathcal{L}_x^O(u) \mathcal{C} \). From the assumption for one-step reduction it follows that there is an unwinding \( u' \) of \( u \) such that \( w'' \in \mathcal{P} \mathcal{L}_x^O(u') \mathcal{C} \). Clearly \( u' \) is an unwinding of \( t \). Vice versa, let the inverse implication holds for one-step unwindings. If the implication holds for \( n \) reduction steps, if \( u \) is a \( n \)-step unwinding of \( t \), and if \( u' \) is a \( n \)-step unwinding of \( u \) and if \( w'' \in \mathcal{P} \mathcal{L}_x^O(u') \mathcal{C} \), then there is a word \( w' \in \mathcal{P} \mathcal{L}_x^O(u) \mathcal{C} \) such that \( w' \Rightarrow_{\mathcal{C}} w'' \). Moreover, there is a word \( w \in \mathcal{P} \mathcal{L}_x^O(t) \mathcal{C} \) such that \( w \Rightarrow_{\mathcal{C}} w' \). Consequently, \( w \Rightarrow_{\mathcal{C}} w'' \).

We prove (1) for one-step unwindings and reductions by structural induction on the complexity of \( t \). (1) clearly holds for variables and constants. Let it hold for all proper subterms of a nonatomic term \( t \). In all cases different from case (3) of the pre-language definition, this equivalence follows directly from the inductive assumption. Let \( t \) have the form \( f(t_1, \ldots, t_k) \). If \( w \in \mathcal{P} \mathcal{L}_x^O(t) \mathcal{C} \) and \( w \Rightarrow_{\mathcal{C}} w' \), then from the definition of \( \mathcal{P} \mathcal{L}_x^O(t) \) it follows that there exists a \( i \) such that \( w \in \mathcal{P} \mathcal{L}_x^O(t_i) \mathcal{C} \). If the word is reduced in the middle, then \( w \) has the form \( w_1 g_{y_i} w_2 f_{y_i} \), for a recursively defined function \( g \) with definition \( g \), and \( w' = w_1 v w_2 f_{y_i} \), for a word \( v \in \mathcal{P} \mathcal{L}_y^O(g) \). From the inductive assumption it follows that \( w_1 v w_2 \) belongs to \( \mathcal{P} \mathcal{L}_x^O(u_1) \) for an unwinding \( u_1 \) of \( t_1 \) and consequently \( w' \in \mathcal{P} \mathcal{L}_x^O(f(t_1, \ldots, u_1, \ldots, t_k)) \). The inverse implication follows in a similar way. It remains to be proven that the equivalence holds in case of topmost unwindings and rightmost reductions. Let \( w \in \mathcal{P} \mathcal{L}_x^O(t) \mathcal{C} \) and let \( w \Rightarrow_{\mathcal{C}} w' \). From the definition of \( \mathcal{P} \mathcal{L}_x^O(f(t_1, \ldots, t_k)) \) it follows that \( w \) has the form \( w_1 f_{y_i} \), for some \( i \), and that \( w_1 \in \mathcal{P} \mathcal{L}_x^O(t_i) \). The rightmost reduction results in \( w_1 v \) for a word \( v \in \mathcal{P} \mathcal{L}_y^O(f) \). Consequently, \( w' = w_1 v \) belongs to the unwinding \( \mathcal{P} \mathcal{L}_x^O(F(t_{1/y_1}, \ldots, t_k/y_k)) \) due to part (5) of the previous lemma 4.1. Vice versa, if \( w' \in \mathcal{P} \mathcal{L}_x^O(F(t_{1/y_1}, \ldots, t_k/y_k)) \), then due to part (5) of lemma 4.1 we can present \( w' \) in the form \( w_1 v \) where \( w_1 \in \mathcal{P} \mathcal{L}_x^O(t_i) \) and \( v \in \mathcal{P} \mathcal{L}_y^O(F) \) and we can define \( w = w_1 f \). In this case \( w \Rightarrow_{\mathcal{C}} w' \) and \( w \in \mathcal{P} \mathcal{L}_x^O(f(t_{1/y_1}, \ldots, t_k/y_k)) \).

We prove (2) by structural induction too. Note that, as in the previous case, it is enough to consider one-step unwindings and reductions. The equivalence clearly holds for atomic terms. Let the equivalence hold for all proper subterms of nonatomic term \( t \). We prove first that the left hand side implies the right hand side. Let \( w \in \mathcal{P} \mathcal{L}_x^C(t) \mathcal{C} \) and let \( w \Rightarrow_{\mathcal{C}} w' \). Since unwinding of a subterm is a subterm of an unwinding of \( t \) and since the closed pre-language of a subterm is included in the closed pre-language of the term (see lemma 4.1, part (2)), we can assume that \( t \) is a minimal term with this property. Since \( t \) is minimal and since it is not atomic, it must have the form \( r \cdot a \) or the form \( f(t_1/y_1, \ldots, t_n/y_n) \) or the form \( t_1 \rightarrow \text{collect}(y \mid t_2) \) due to lemma 4.1, part (3). In the first case \( w \in \mathcal{P} \mathcal{L}_x^O(t) \mathcal{C} \) and we can present \( w \) in the form \( w a \mathcal{C} \) where \( v \in \mathcal{P} \mathcal{L}_x^O(r) \). Since the
reduction rules cannot be applied to $a^C$, we can present $w'$ in the form $v'a^C$. From (1) it follows that there is an unwinding $r'$ of $r$ such that $v' \in \text{Pre}L^O_x(x')$. Consequently, $w' = v'a^C \in \text{Pre}L^O_x(r', a)^C$ and we can define $u$ to be equal to $r', a$. Let $t$ have the form $f(t_1/y_1, \ldots, t_n/y_n)$. Due to the minimality assumption and the definition of open pre-languages, we can present $w$ in the form $v_t^C$ where $v \in \text{Pre}L^O_x(t_1)$. If the reduction is applied to a nonterminal in $v$ then the proof follows as in the previous case. If it is applied to the rightmost position, then there is a word $v_1 \in \text{Pre}L^C_y(F)^C$ such that $w' = vv_1$. From lemma 4.1, part (6) and from the definition of operation $C$ it follows that $vv_1 \in \text{Pre}L^C_x(F(t_1/y_1, \ldots, t_n/y_n))^C$. Let $t$ have the form $t_1 -> \text{collect}(y \mid t_2)$ and let $w \in \text{Pre}L^C_x(t)^C$. From the minimality assumption it follows that $w$ can be decomposed into $w_1 \in \text{Pre}L^O_x(t_1)$ and $w_2 \in \text{Pre}L^C_y(t_2)^C$. If $w_1$ is reduced, then the implication follows from part (1) and lemma 4.1, part (6). In the other case it follows from the inductive assumption; i.e., there is a one-step unwinding $u_2$ of $t_2$ such that $w_2' \in \text{Pre}L^C_y(u_2)$, where $w' = w_1w_2'$. Consequently, $w_1w_2' \in \text{Pre}L^C_x(t_1 -> \text{collect}(y \mid u_2))^C$ due to lemma 4.1, part (6) and to the definition of the closed pre-language, part (5).

Now we prove the inverse implication. It is enough to prove that if $u$ is one-step unwinding of $t$ and $w' \in \text{Pre}L^O_x(u)^C$, then there exists a word $w$ such that $w \in \text{Pre}L^C_x(t)^C$ and $w \Rightarrow_C w'$ or $w = w'$. This implication clearly holds for atomic terms. Let $u$ be obtained from $t$ by replacing a function symbol $f$ by its definition $F$, and let $w' \in \text{Pre}L^C_x(u)^C$. There are two possible cases: either $u = r[F(t/y)/z]$, for a nonvariable term $r$, or $u = F(t/y)$. Let $u$ have the form $r[F(t/y)/z]$. If $w'$ belongs to a proper subterm of $u$, then the implication follows from the inductive assumption and lemma 4.1, part (2), since we can take a proper subterm $s$ of $t$ and its unwinding $s'$ which is a proper subterm of $u$. If not, then the implication follows from lemma 4.1, part (3) and from the inductive assumption.

Let $u$ have the form $F(t/y)$. If $u$ is minimal such that $w' \in \text{Pre}L^C_x(u)^C$, then from lemma 4.1, part (3) it follows that either $w' \in \text{Pre}L^C_x(u)^C$, or $t$ has the form $t_1 -> \text{collect}(y \mid t_2)$ and $w$ can be decomposed into $w_1 \in \text{Pre}L^O_x(t_1)$ and $w_2 \in \text{Pre}L^C_y(t_2)^C$. In the first case, the implication follows from (1) and lemma 4.1 part(3). In the second case, if the unwinding is applied to $t_1$, then the implication follows from (1). If it is applied to $t_2$, then the implication follows from the inductive assumption and lemma 4.1, part (6).

Let $u$ be not minimal and let $s$ be a minimal proper subterm of $u$ such that $w' \in \text{Pre}L^C_x(s)^C$. If there is an $i$ such that $s$ is a subterm of $t_i$, then we can define $w = w_i$. In this case $w = w' \in \text{Pre}L^C_x(u) \cap \text{Pre}L^C_x(t)$. In the other case $F$ can be presented in the form $F_0[s_0/z]$ and $s$ can be presented in the form $s_0(t/y)$. From lemma 4.1, part (3) and part (2) it follows that $w' \in \text{Pre}L^O_x(s)^C$, or $s$ has the form $s_1 -> \text{collect}(y \mid s_2)$ and $w'$ can be decomposed into $w_1w_2$ so that $w_1 \in \text{Pre}L^O_x(s_1)$ and $w_2 \in \text{Pre}L^C_x(s_2)^C$, where $|w_2| > 0$.

If $w' \in \text{Pre}L^C_x(s)^C$, then we can decompose $w'$ into $w_1 \in \text{Pre}L^O_x(t_1)$ and $w_2 \in \text{Pre}L^O_y(s_0)^C$. Since $s_0$ is a subterm of $F$ and since $\text{Pre}L^C_y(s_0)^C \subseteq \text{Pre}L^C_y(F)^C$, due to lemma 4.1 part (2), we can define $w$ to be equal $w_1w_2^C$. Clearly $w \in \text{Pre}L^C_x(t)$ since $f(t/y) = t$. Technical Report: On the Validation of Invariants at Runtime –
If \( s \) has the form \( s_1 \rightarrow \text{collect}(y \mid s_2) \), then similarly \( s_0 \) has the form \( s_{01} \rightarrow \text{collect}(y \mid s_{02}) \) and we can present \( s_j \) in the form \( s_{0j}(t/y) \), for \( j = 1, 2 \) (see Figure 7). Note that the terms in the tuple \( z \) are not substituted for the bound variable \( y \) in \( F \). Since \( w' \in \mathcal{P}_{\mathcal{L}_x}(s) \) and since \( s \) is minimal, we can decompose \( w' \) into \( w_1w_2w_3 \) where \( w_1 \in \mathcal{P}_{\mathcal{L}_x}(t_i) \), for some \( i \), \( w_2 \in \mathcal{P}_{\mathcal{L}_{y_1}}(s_{01}) \) and \( w_3 \in \mathcal{P}_{\mathcal{L}_{y_2}}(s_{02}) \). From the definition of the closed pre-language it follows that \( w_2w_3 \in \mathcal{P}_{\mathcal{L}_{y_1}}(F) \). We can define \( w \) to be equal \( w = w_1f_{y_i}^C \).

We associate a context free grammar \( G_x(t) = \{ T, Nt, s_x, \rightarrow \} \) with an OCL-term \( t \) and its free variable \( x \). We do not adorn \( \rightarrow \) and \( Nt \) with \( t \) and \( x \) since the term and the variable will be clear from the context. This reduction relation differs from \( \rightarrow^C \) only in that we add rules for the start symbol. The set of terminals \( T \) is defined in subsection 4.2. The set of nonterminals \( Nt \) contains the set \( Nt_0 \) and includes the nonterminal \( s_x, i.e., Nt = Nt_0 \cup \{ s_x \} \). Reduction relation \( \rightarrow \) includes all reductions of the form \( \rightarrow^C \), reductions of the form \( s_x \rightarrow \mathcal{P}_{\mathcal{L}_x}(t)^C \), of the form \( e^C \rightarrow e \) and \( a^C \rightarrow a \), for every nonterminal \( a^C \) corresponding to an object-property. We denote the language generated by the grammar \( G_x(t) \) by \( L(G_x(t)) \) or simply by \( L_x(t) \).

**Corollary 4.1.** Let \( t \) be an OCL-term of the restricted syntax and let \( x \) be a free variable of \( t \). Then the following conditions hold:

1. \( s_{xt} \Rightarrow^* A_1 \ldots A_{n-1}A_n^C \) if, and only if, there exists an unwinding \( u \) of the term \( t \) such that the partial word \( A_1 \ldots A_{n-1}A_n^C \) belongs to \( \mathcal{P}_{\mathcal{L}_x}(u)^C \)

2. If \( s_{xt} \Rightarrow^* A_1 \ldots A_n \), then \( s_{xt} \Rightarrow^* A_1 \ldots A_{i-1}A_i^C \), for \( i \leq n \)

3. The language \( L_x(t) \) is prefix-closed

**Proof:**
(1) follows directly from the definition of \( \rightarrow \), the Reduction Lemma 4.2, part (2) and the fact that a
The reduction relation \( \rightarrow \) cannot be applied in a derivation of a word of the form \( w^C \) since there is no inverse rule allowing to obtain nonterminal \( a^C \) at the end.

We prove (2). If \( s_{xt} \Rightarrow^* A_1 \ldots A_{n-1} A_n \), then \( s_{xt} \Rightarrow^* A_1 \ldots A_{n-1} A_n^C \) since we can avoid applying rules of the form \( a^C \rightarrow a \). From (1) it follows that there exists an unwinding \( u \) such that \( A_1 \ldots A_{n-1} A_n^C \in \text{Pre}_x L^C_x(u)^C \). From lemma 4.1, part (7) it follows that \( A_1 \ldots A_t \in \text{Pre}_x L^C_x(u)^C \), and consequently, \( A_1 \ldots A_0^C \in \text{Pre}_x L^C_x(u)^C \). From (1) it follows that \( s_{xt} \Rightarrow^* A_1 \ldots A_n^C \).

The condition (3) follows from the condition (2), since if \( s_{xt} \Rightarrow^* a_1 \ldots a_n \), then there is a reduction of the form: \( s_{xt} \Rightarrow^* a_1 \ldots a_n^C \Rightarrow a_1 \ldots a_i \).

As mentioned at the beginning of this subsection, for every context free language \( L \), the language \( L/a = \{ w | wa \in L \} \) is context free as well. We show now how to transform \( G_x(t) \) to a context free grammar corresponding to \( L(G_x(t)/a) \). We denote the transformed grammar by \( G_x(t)/a \). We present such a grammar in the form \( (T, NT/a, s_{xt}/a, \gamma_a) \), where \( NT/a = Nt \setminus \{s_{xt}\} \cup \{s_{xt}/a\} \). The reduction relation \( \rightarrow \gamma_a \) is obtained from \( \rightarrow \) by replacing every rule of the form \( s_{xt} \rightarrow W \) with rule \( s_{xt}/a \rightarrow \gamma_a W \), by removing reductions rule of the form \( c^C \rightarrow c \) and \( b^C \rightarrow b \) for \( b \neq a \), and by replacing rule \( a^C \rightarrow a \) with rule \( a^C \rightarrow \gamma_a \epsilon \). All other rules remain unchanged; i.e., the rules have the form: \( s_{xt}/a \rightarrow \gamma_a \text{Pre}_y L^C_y(t)^C, f_{i,j} \rightarrow \gamma_a \text{Pre}_y L^C_y(F_1)^C \) and \( a^C \rightarrow \gamma_a \epsilon \).

**Lemma 4.3.** If \( t \) is an OCL-term of the restricted syntax, then \( L(G_x(t)/a) = L_x(t)/a \).

**Proof:**
Let \( w \) belong to \( L_x(t)/a \). Then \( wa \) belongs to \( L_x(t) \) and there is a derivation of the form \( s_{xt} \Rightarrow^* wa^C \Rightarrow wa \). The rules removed from \( \rightarrow \) concern only nonterminals of the form \( c^C \) and \( b^C \); such nonterminals can occur only at the end of a derivable word and \( \rightarrow \) allows us to reduce those nonterminals only to \( c \) and \( b \), respectively. Consequently, \( s_{xt}/a \Rightarrow^* \gamma_a \) \( wa^C \Rightarrow^* \gamma_a \) \( w \). Vice versa, if a word \( w \) belongs to \( L(G_x(t)/a) \), then there is a derivation of the form \( s_{xt}/a \Rightarrow^* a_n \ldots A_0^C \Rightarrow^* a_n \ldots A_0^C = \gamma_a wA^C \Rightarrow^* \gamma_a \) \( w \) since we can reduce nonterminal \( A^C \) at the end of the reduction process. The only way of removing nonterminals of the form \( A^C \) is by applying the rule \( a^C \rightarrow \epsilon \). All other rules retain the nonterminal at the end. Thus, \( A^C = a^C \).

### 4.5. Type Correctness

In this subsection we deal with the question of type correctness of OCL-terms corresponding to partial words. We prove that such terms are type correct in respect to the type system defined in subsection 4.1. In the following we assume that there is an underlying class model defining a number of class types and an inheritance relation between them.

Let \( a \) be an object-property of class \( A \). The corresponding nonterminal \( a^C \) is formalized simply by variable \( x : A \); this is due to the fact that in the grammar corresponding to \( \rightarrow \gamma_a, a^C \) is reduced to \( \epsilon \) (see subsection 4.4). For a recursively defined function symbol \( f(y_1 : C_1, \ldots, y_n : C_n) \) we defined nonterminals \( f_{y_1} \). Here we define a new function symbol \( (x : C_j) . f_{y_1} : \text{Collection}(T) \), where \( T \) is the type of \( f \); i.e., we assume that the function has one argument.
Due to points (2), (3) and (4) of the next lemma we can eliminate a nonterminal \( f_{y_1} \) corresponding to a function \( f \), if \( f \) is not of a class type; similarly we can eliminate nonterminals \( f_{y_1} \) and \( f_{y_1}^C \) if the variable \( y_1 \) is not of a class type. Consequently we can remove reduction rules including such nonterminals without changing expressivity of the resulting grammar. Similarly, we can eliminate words of the form \( wc^C \) and \( wb^C \), where \( b \) is a property different from \( a \) since they cannot be reduced to total words using \( \rightarrow_a \).

In general, the elimination is possible due to the assumption that properties have arguments of a class type, and consequently navigation via BOT types and \( \text{OCLAny} \) is disallowed. On the other hand without this elimination, some well typed terms would result with words which do not correspond to well typed terms. For example let \( f(x : \text{Integer}) \) be a function symbol and let property \( a \) have values of a class type. Term \( f(\text{self}.a->\text{size}()) \) is well typed but results with word \( af_x \), which does not correspond to a well typed term. Note that we do not eliminate nonterminals \( f_{y_1}^C \) in case when \( f \) is not of a class type since the words corresponding to \( f_{y_1}^C \) correspond to navigation to property \( a \), which by assumption has arguments of a class type.

In the following, we say that a nonterminal \( f_{y_1} \) \( (f_{y_1}^C, \text{resp.}) \) corresponds to argument of a class type, if \( y_1 \) is of a class type. Similarly we say that \( f_{y_1} \) corresponds to values of a class type, if \( (y_1 : C_1).f_{y_1} \) is of a class type. As mentioned above \( f_{y_1}^C \) corresponds by definition to values of a class type since \( (y_1 : C_1).f_{y_1}^C \) is of a class type.

Point (5) and (6) of the next lemma say that OCL-terms formalizing words extracted from other OCL-terms are well typed.

**Lemma 4.4.** Let \( t(x : T,...) \) be a well typed OCL-term. Let \( a \) be an object-property of a class \( A \). Let \( f(y_1 : T_1, \ldots, y_n : T_n) \) be a function.

1. Let \( T \) be BOT or \( \text{OCLAny} \). Then \( \text{PreL}_x^O(t) \) does not contain total words different from \( \epsilon \). Moreover if \( t \) is of type \( \text{Collection}(C) \) for a class \( C \), then \( \text{PreL}_x^O(t) \) does not contain total words

2. Let \( f \) be of type \( \text{Collection}(C) \), for a class \( C \), and let \( T_1 \) be a BOT or \( \text{OCLAny} \). If \( f_{y_1} \Rightarrow_a^* w \), then \( w \) is not a total word

3. Let \( T_1 \) be a BOT or \( \text{OCLAny} \). If \( f_{y_1}^C \Rightarrow_a^* w \), then \( w \) is not a total word

4. Let \( s_{xT/a} \Rightarrow_a^* w \) and let \( w \) contain a nonterminal \( f_{y_1} \) corresponding to arguments not of a class type or values not of a class type, then \( w \) cannot be reduced to a total word

5. If \( t \) is of type \( \text{Collection}(C) \) for a class \( C \), \( w \in \text{PreL}_x^O(t) \) and every nonterminal occurring in \( w \) corresponds to arguments and values of class types, then term \( (x : T).w \) corresponding to \( w \) is well typed and has type \( \text{Collection}(C) \)
6. If \( w = v^a C \in \text{Pre} L^O_x (t)^ C \) and all nonterminals in \( v \) correspond to arguments and values of class types, then the term \( (x : T).v \) corresponding to \( v \) is well typed and \( (x : T).v \) has type Collection(A).

\textbf{Proof:}

We prove the second part of (1) by contradiction. Clearly \( \text{Pre} L^O_x (t) \) is empty if \( t \) is a constant. Term \( t \) cannot be a variable, since type OCLAny and the BOTs are not subtypes of Collection(C), for a class C. Let \( t \) be the smallest term for which this property does not hold. Since \( t \) is of type Collection(C) for a class C, its topmost symbol must be if then else endif, union, Set\{r_1, ..., r_n\}, a recursively defined function symbol \( f \), an object-property \( a \) or collect. The first three cases cannot hold due to the minimality assumption. In the fourth case by definition, there are no total words in the pre-language. In the fifth case, \( t \) has the form \( r . a \), for well typed term \( r, w = w_1 a \) and \( w_1 \in \text{Pre} L^O_x (r) \). Since an object-property can have only arguments of a class type or the corresponding collection type, \( r \) is of type Collection(D), for a class D. But this contradicts the minimality assumption since \( r \) is a proper subterm of \( t \). In the last case, \( t \) has the form \( t_1 \rightarrow \text{collect}(y \mid t_2), w = w_1 w_2 \in \text{Pre} L^O_x (t_1) \) and \( w_2 \in \text{Pre} L^C_y (t_2) \) according to the definition of closed pre language. If \( y \) is of class type, then \( t_1 \) contradicts the minimality assumption; if not, then \( t_2 \) does.

We prove the first part of (1) by contradiction too. Let \( t \) be a minimal term such that this property does not hold. \( t \) cannot be atomic. From the minimality assumption and the fact that \( \text{Pre} L^O_x (t) \) contains a total word it follows that \( t \) must be of the form \( r . b \) or \( t_1 \rightarrow \text{collect}(y \mid t_2), w = w_1 w_2 \) and \( w_1 \in \text{Pre} L^O_x (t_1) \) and \( w_2 \in \text{Pre} L^O_y (t_2) \). In the first case, there is a contradiction with the property proven above since \( r \) must be of a class type. In the second case, if \( y \) is of a class type, then \( t_1 \) contradicts the property proven above. If not, then \( |w_1| > 0 \) and \( t_1 \) contradicts the minimality assumption, or \( |w_2| > 0 \) and \( t_2 \) does.

(2) follows from (1), the Reduction Lemma 4.2 and the definition of \( \rightarrow_a \).

To prove (3) it is enough to prove that if \( x \) is of type BOT or OCLAny, then \( \text{Pre} L^C_x (t) \) does not contain total words different from \( \epsilon \) since \( f^C_{yi} \Rightarrow^* \epsilon \) \( w \) if, and only if, \( f^C_{yi} \Rightarrow \epsilon \) \( w_{ai} \) if, and only if, \( wa \in \text{Pre} L^C_y (u) \) for an unwinding \( u \) of \( F \), where \( F \) is the definition of \( f \). Suppose the opposite and let \( u \) be a minimal term such that a total word \( wa \) belongs to \( \text{Pre} L^C_y (u) \). From lemma 4.1, part (3) and the fact that \( wa \) is a total word it follows that \( u \) has the form \( r . a \) and \( wa \in \text{Pre} L^O_y (r . a) \), or the form \( t_1 \rightarrow \text{collect}(y \mid t_2), w = w_1 w_2, |w_2| > 1, w_1 \in \text{Pre} L^O_x (t_1) \) and \( w_2 \in \text{Pre} L^C_y (t_2) \). In first case there is a contradiction with the second part of (1), since \( r \) is of a class type and \( w \in \text{Pre} L^O_x (r) \). In the second case, if \( y \) is of a class type, then there is a contradiction with (1); if not, then \( t_2 \) contradicts the minimality assumption.

To prove (4), we show the following property first: if \( t \) is of type Collection(C) for a class C, \( w \in \text{Pre} L^O_x (t) \) and \( w \) contains a nonterminal \( f^C_{yi} \) corresponding to arguments or values not of a class type, then \( w \) cannot be reduced to a total word using \( \rightarrow_C \). Assume the opposite and let \( t \) be a minimal term in respect to the number of contained operations such that \( w \in \text{Pre} L^O_x (t) \), \( w \) contains \( f^C_{yi} \) corresponding to arguments or values not of a class type and \( w \) can be reduced to a total word. It is easy to show that at the top of this term only a property \( b \), recursively defined symbol \( g \) or a collect can occur. In the first case, \( w = w_1 b \), \( t \) has the form \( r . b \) and \( w_1 \in \text{Pre} L^O_x (r) \). Due to the assumption that object-properties have arguments of a class type, and the fact that \( f^C_{yi} \) has to occur in \( w_1 \) we get a...
contradiction with the minimality assumption. If the topmost operation is \( g \), then \( w = w_1g_{yj} \) and \( y_j \) must be of a class type since in the other case \( g_{yj} \) would not be reducible to a total word, as shown in (2). Nonterminal \( g_{yj} \) corresponds to values of class type, and therefore it is different from \( f_{nj} \). However, this contradicts the minimality assumption as in the previous case. Thus, the only possibility is that \( t \) has the form \( t_1 \rightarrow \text{collect}(y \mid t_2) \), \( w = w_1w_2 \), \( w_1 \in \mathcal{P} \mathcal{L}_x^O(t_1) \) and \( w_2 \in \mathcal{P} \mathcal{L}_x^O(t_2) \). \( t_2 \) has to be of type \( \text{Collection}(C) \). If \( w_2 \) contains \( f_{yj}^{n_i} \), then there is a contradiction with minimality assumption. Let \( w_1 \) contain \( f_{yj}^{n_i} \), \( y \) must be of a class type since in the other case for every unwinding \( u_2 \) of \( t_2 \), \( \mathcal{P} \mathcal{L}_x^O(t_2) \) would not contain total words, as proved in (1), and consequently \( w_2 \) would not be reducible to a total word. This means that \( t_1 \) has to be of a class type, again a contradiction with the minimality assumption.

We prove (4) by contradiction. Let there be a partial word \( w \) including a nonterminal \( f_{yj} \) corresponding to arguments or values not of a class type such that \( s_{\times a} \Rightarrow \gamma_a^* \) \( w \) and \( w \) is reducible to a total word using \( \rightarrow \gamma_a \). If \( w \) does not contain a nonterminal of the form \( a^C \) at the end, then at some point in the process of obtaining \( w \) nonterminal \( a^C \) was removed. We can postpone the reduction of \( a^C \) and deal with words of the form \( wa^C \) and unwindings of \( t \). Let \( t' \) be an unwinding of \( t \) such that \( wa^C \in \mathcal{P} \mathcal{L}_x^O(t')^C \) and let \( wa^C \). There must be such a term due to the Reduction Lemma. Consequently, it is enough to prove that partial word \( wa \in \mathcal{P} \mathcal{L}_x^O(t') \) cannot be reduced to a total one. Points (3) and (4) of lemma 4.1 imply that either \( r.a \) is subterm of \( t' \) and \( wa \in \mathcal{P} \mathcal{L}_x^O(r.a) \), or there is a decomposition of a subterm \( u \) of \( t' \) as described in point (4). In the first case, there is a contradiction with the property proved above since \( r \) must be of a class type. In the other case, the subterm can be decomposed as described in point (4) of lemma 4.1; we can assume that \( u_n \) is of the form \( r.a, w_n = va \) and \( v \in \mathcal{P} \mathcal{L}_x^O(r) \). If \( v \) contains the improper nonterminal, then there is a contradiction with the minimality assumption. If not, then it must be contained in a partial word \( w_i \), for some \( i < n \). \( v \) is reducible to a total word, therefore there is an unwinding \( r^i \) of \( r \) containing a total word. Due to the fact that \( r \) is of a class type, \( r^i \) is of the class type too. (1) implies that \( x_n \) is of a class type too. Similarly since \( w_j \in \mathcal{P} \mathcal{L}_x^O(u_j) \), \( x_j \) must be of a class type, for all \( j < n \); in particular \( x_i \) must be of a class type. But then \( u_1 \) must be of a class type. This is contradiction with the property proven above since \( w_i \in \mathcal{P} \mathcal{L}_x^O(u_i) \) and \( w_i \) is reducible to a total word.

We prove (5) by structural induction. Let its assumption be satisfied. (5) clearly holds if \( t \) is an atomic term. Let it holds for all proper subterms of \( t \) and let \( w \in \mathcal{P} \mathcal{L}_x^O(t) \). Due to the type of \( t \), the list of topmost symbols is restricted. If the topmost symbol is if then else endif, union or \( \text{Set}\{r_1, \ldots, r_n\} \), then \( t \) has a subterm of type \( \text{Collection}(C) \) and the property follows from inductive assumption (see the typing rules in subsection 4.1). If \( t \) has the form \( r.b \), for an object-property \( b \), then \( r \) is of type \( \text{Collection}(D) \), for a class \( D, w = w_1b \) and \( w_1 \in \mathcal{P} \mathcal{L}_x^O(r) \). Moreover, \( b : D \rightarrow \text{Collection}(C) \). From the inductive assumption it follows that \( (x : T).w_1 \) has type \( \text{Collection}(D) \). Due to the lifting property, \( b \) can be applied to \( \text{Collection}(D) \). Consequently, \( (x : T).w \) is well typed and has type \( \text{Collection}(C) \). If the topmost symbol is a recursively defined function symbol \( f \) of type \( \text{Collection}(C) \) and if variable \( y_j \) is of type \( D \) for a class \( D \), then due to inductive assumption \( w = w_1f_{yj} \), the term \( w_1(x : T) \) is well typed and has type \( \text{Collection}(D) \). Consequently, \( (x : T).w \) is well typed and has type \( \text{Collection}(C) \). Before we deal with case of collection, we note that if \( t \) is of a class type, \( w \in \mathcal{P} \mathcal{L}_x^O(t(x : T, \ldots)) \) and \( w \) does not
include nonterminals corresponding to arguments or values not of class types, then $T$ must be of a class type (we skip the easy proof by structural induction). In case of $t_1 \Rightarrow \text{collect}(y \mid t_2)$, $t_2$ must be of the same class type as $t$, i.e., $\text{Collection}(C)$. The above-mentioned property implies that $y$ and consequently $t_1$ must be of a class type $B$. If $w_1 \in PreL^O_C(t_1)$ and $w_2 \in PreL^O_C(t_2)$, then from the inductive assumption follows that, $(x : T).w_1$ is of type $\text{Collection}(B)$ and $(y : B).w_2$ is of type $\text{Collection}(C)$. The term $(y : B).w_2[(x : T).w_1/y]$ is well typed and has type $\text{Collection}(C)$.

Let the assumptions of (6) be satisfied. We prove (6) by structural induction. It clearly holds for atomic terms. Let it hold for all subterms of term $t$. Let $w \in PreL^C_C(t)$; we can assume that $t$ is a minimal with this property. From lemma 4.1, part (3) and the minimality assumption follows that $t$ has the form $r.a$ and $w \in PreL^O_C(t)$, or the topmost operation in $t$ is collect, the term and the partial word can be decomposed as described in lemma 4.1, part (4); in particular $w = w_1 \ldots w_n$ and $w_i \in PreL^O_C(u_i)$, for $i = 2, \ldots, n$. Term $t$ cannot have the form $f(t_1, \ldots, t_n)$. If $t$ has the form $r.a$, then $r$ is of type $\text{Collection}(A)$ since $a$ has arguments of type $A$. $w = va^C$ and $v \in PreL^O_C(r)$. From (5) it follows that $(x : T).v$ is of type $\text{Collection}(A)$. Consequently, $(x : T).v$ is well typed and $(x : T).v$ is of this type too. In case of collect, we can assume that $u_a$ has the form $r.a$ and $w_n = w'a$ since we can assume that $u_a$ is a minimal subterm of $t_n$ and make the decomposition of collect terms. $r$ must be of type $\text{Collection}(A)$. Since $w_i \in PreL^O_C(u_i)$, the property mentioned in the proof of (5) implies that $x_n$ must be of a class type. As in proof of (5) we can conclude that $x_i$ is of a class type $C_i$ and that $w_{i-1}$ is of class type $\text{Collection}(C_i)$, for $i = 2, \ldots, n$. The typing rule for collect and the lifting property imply that the term $v[w_{n-1}/x_n]\ldots[w_1/x_2]$ is well typed. Therefore, term $v[w_{n-1}/x_n]\ldots[w_1/x_2]$ corresponding to $v$ is well typed too and has type $\text{Collection}(A)$.

The previous lemma implies that in the definition of indistinguishability we can consider only indistinguishability at objects (see subsection 5.2), since for values of basic OCL types the corresponding languages are empty. The following corollary says that OCL-terms resulting from the right hand sides of the rules corresponding to recursively defined function symbols are well typed and have the same type as the function symbols. It says also that all partial words derivable form the start symbol correspond to well typed OCL-terms.

**Corollary 4.2.** Let $a$ be an object-property of class $A$ and let $F$ be the definition of function symbol $f$.

1. If $s_{xT}a \Rightarrow y_a$, nonterminals in $w$ correspond to arguments and values of class types, then the OCL-term $(x : T).w$ corresponding to $w$ is well typed and has type $\text{Collection}(A)$

2. Let $t$ be a type correct OCL-term of the restricted syntax, let $w$ be an OCL-term corresponding to a partial word $w \in PreL^C_C(t)/a$ and let the nonterminals occurring in $w$ correspond to argument and values of class types. Then the OCL-term $w^{rev}$ corresponding to the reversed word (see section 4) is type correct

**Proof:**

(1) follows from part (6) of the previous lemma and from part (2) of lemma (4.2). Part (2) of this
lemma follows easily from part (1) and from part (5) of lemma (4.4) by induction on the length of partial words.

5. Invariant Validity

The evaluation of an OCL-term means navigation through a network of objects in order to determine the value of the term. We show below that the value of a term is determined by the object network defined by the context free language corresponding to the term. We prove that for two system states, the value of a term is the same if the network defined by the term is the same in both states. Consequently, such a network is an upper bound of its footprint (cf. [7]). Our method allows us to detect the, what we call, roots of networks which are modified by a method execution. In subsection 4.5, we showed that total words in the extracted languages correspond to well typed OCL-terms (see lemma 4.4). Those terms have the property that an object is defined by such a term if, and only if, there is a word in the corresponding language marking a path from a network root to that object. If a network of objects is modified, then we can use the inverse language, or a properly defined OCL-term corresponding to the inverse language, to navigate to the root of the modified network. An invariant concerns all objects of the corresponding class. A brute force validation method requires checking of the invariant for all objects of the corresponding class. Our method reduces the search space to the roots of modified networks.

5.1. Semantics

In this section we present a formal semantics of OCL. There are a number of other OCL semantics (see the references in Cengarle and Knapp [11]). The semantics presented in this paper is a slight modification of the semantics defined by Bidoit et al. [10] (see also [27]). The OCL semantics defined in the papers [11, 10] strictly differentiate between UML queries and state-changing operations. Queries are part of the functional signature; whereas non-query operations are treated as relations which for a given system state output a new system state and optionally a value. Most formal semantics of OCL interpret classes as sets. For example, the semantics proposed in [10] interprets C.allInstances() as the set of all objects of class C which exist at a given moment of time. In this semantics, one can define C.allInstances@pre() as the set of objects existing at the moment when a method is invoked.

We use italics to distinguish a formal term from its OCL counterpart. For a class model CM, we define a class signature \( \Sigma \) of the form \((S, \leq, F)\) (cf. Goguen and Meseguer [16]). \( S \) is a set of sorts. It includes sorts Boolean, Real and Integer, for the basic OCL types, a sort \( C \), for every class \( C \) in the class model, and the sort OclAny. For those sorts it includes also the corresponding collections and for OclAny. Moreover, we assume that there is a sort State corresponding to global system states. The set of sorts is partially ordered by the relation \( \leq \) mirroring the subclass and subtype relation, i.e., \( B \) is a subclass or a subtype of \( A \) if, and only if, \( B \leq A \).

\( F \) is a set of typed function symbols corresponding to properties occurring in the class model and to predefined OCL operators, such as includes. In OCL-terms, the state parameter is hidden, but
occurs explicitly in theirs formalizations. It corresponds to the current state of the underlying system. For example, if a is an attribute of class C with values of type T, then $\sigma$ contains function symbol $a : \text{State} \times C \rightarrow T$. The value of a can change as the system evolves. Consequently, its value depends on the actual parameter of class C and the current state. A class attribute D.d with values of type T is formalized by function $D.d : \text{State} \rightarrow T$. Similarly, the feature $D.allInstances()$, defining all existing object of class D, is formalized by function $D.allInstances : \text{State} \rightarrow \text{Collection}(D)$. The OCL function $\text{includes}$ is formalized by function $\text{includes} : \text{Collection}(T) \times T \rightarrow \text{Boolean}$. Note that we do not need to know the state to figure out whether an object belongs to a set or not.

If a is an association-end leading from class A to class B, b is an association-end leading from B to C, then the OCL-term $\text{self}_A.a.b$ is a well typed. Those associations are formalized by terms $(s : \text{State}, \text{self}_A)a$ and $(s : \text{State}, \text{self}_B)a$, respectively. The OCL-term is formalized by term $(s : \text{State}, (s : \text{State}, \text{self}_A)a)b$. Observe that the state variable s occurs twice. This is due to the fact that the navigation through associations and the attribute evaluation do not change system states (cf. [11, 38]).

An OCL-term of the form $\text{self} . a_1 \ldots a_n$ can be represented as a word $a_1 \ldots a_n$ and as a formal term $(s : \text{State}, \ldots (s : \text{State}, \text{self} : C)a_1 \ldots )a_n$. In case of variable $\text{self}$, we use an equally named formal variable $\text{self}$. Similarly, for a word $a_1 \ldots a_n \in \mathcal{PC}_{\text{self}}(t)$ we define the corresponding OCL-term $\text{self} . a_1 \ldots a_n$ and its formalization $(s : \text{State}, \ldots (s : \text{State}, \text{self})a_1 \ldots )a_n$. The terms are considered in the corresponding context, i.e., $\text{self}$ is considered to have the C.

We identify singleton sets with their unique elements as it is done in some set theories (cf. e.g. Quine [42]). This allows us to lift functions to sets of values and simplifies the treatment of collections (see subsection 4.1). For a set A, we define the corresponding collection $\text{collection}(A)$ as the set containing all finite set-theoretic unions of sets in A. For example, for the set $\{a, b, c\}$ as well as the set $\{a, b, c, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$, the corresponding collection has the form $\{\emptyset, a, b, c, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$.

As mentioned above, we identify for example $\{a\}$ with a and $\perp$ with $\perp$. This kind of construction corresponds to so called commutative monoids.

Formal models corresponding to $\Sigma$-signatures are defined in a standard way. A model $M$ corresponding to a class signature $\Sigma$ is called $\Sigma$-model. For an OCL type T, T and $T^M$ denote the sort and the set corresponding to the sort T, respectively. We assume that for every sort T different from $\text{State}$, the undefined element $\perp$ belongs to $T^M$. Sorts are interpreted by sets. Moreover, if $T^M$ is the interpretation of T in model M, then $\text{Collection}(T)$ is interpreted as $\text{collection}(T^M)$; in particular, if $T^M$ includes $\perp$, then $\text{collection}(T^M)$ contains $\perp$ as well. For a function symbol f, $f^M$ denotes the interpretation of f in the model M. The set $T.allInstances^M(\sigma)$ includes all objects of class or type T which exist in state $\sigma$; to make the notation more compact we write $T^M_{\sigma}$ instead. Thus, for a state $\sigma$, $T^M_{\sigma} \subseteq T^M$. Inheritance corresponds to the sub-sort relation and set inclusion. If $T_1$ subclasses or subtypes $T_2$, then the corresponding sorts are ordered $T_1 \leq T_2$ and the set inclusion $T^M_1 \subseteq T^M_2$ must hold. In this case, we assume also that $T^M_{\sigma} = T^M_{\sigma} \cap T^M_{\sigma}$. Note that for a class C, $C^M \subseteq \text{collection}(C^M)$ since we identify set elements with the corresponding singletons. In case of predefined OCL types such as Real, we assume that $\text{Real}^M$ is the set of all real numbers and that the set of reals is immutable, i.e., the equation $\text{Real}^M = \text{Real}^M$ holds for every state $\sigma$ (cf. [10]).

For an object-property a with arguments of class C and values of type T, a state $\sigma$, and an object o which exists in state $\sigma$ (i.e., $o \in C_\sigma$), we require that the value exists in this state as well, i.e.,
The value of term \( t \) has value \( \bot \) if, and only if, there is an unwinding \( u \) of \( f_i(x_1, \ldots, x_n) \) such that \( v^{M_i+u} = a \). If there is no such a value and no such an unwinding, then \( f_i(x_1, \ldots, x_n)^M v = \bot \).

5.2. Validity of Invariant versus Networks of Objects

In this subsection, we present the main theorem. It relates the preservation of invariant validity and the presentation of objects’ networks. We use here the OCL semantics outlined in the previous subsection.

It should be noted that for every total word \( w = a_1 \ldots a_n \) from a language corresponding to an OCL-term, we can define in a natural way a well typed OCL-term \( \text{self}.a_1 \ldots a_n \) corresponding to the path defined by the word \( w \) and a formalization \( w = (\sigma, \ldots (\sigma, \text{self})a_1 \ldots) a_n \) of \( w \) (see subsection 4.5).

Definition 5.1. Let \( L \) be a prefix-closed, partial language, let \( M \) be a model, let \( \sigma, \sigma' \) be two states and let \( o_p \) be an object of class \( A \) which exists in both states or a value of BOT. We say that those states are indistinguishable in respect to \( L \) at \( o_p \) if, and only if, the following conditions are satisfied:
1. For every total word \( w \in L \), the corresponding OCL term \( w \) is type correct (cf. subsection 4.5).

2. For every total word \( w \in L \), the value of the corresponding formal term \( w(s, x) \) in state \( \sigma \) is equal to the value of the term in the state \( \sigma' \), i.e., \( M[s \mapsto \sigma, x \mapsto o_p] = M[s \mapsto \sigma', x \mapsto o_p] \).

3. If a total word \( wa \) belongs to \( L \), then for the formal term \( w \) corresponding to \( w \) and for every \( o \in M[s \mapsto \sigma, x \mapsto o_p], (\sigma, o)M = (\sigma', o)M \).

In other words, two states are indistinguishable at an object in respect to a language \( L \), if terms formalizing words of \( L \) are well-typed, the values of those terms are equal and for any proper prefix of a word in \( L \), and the corresponding term, the suffixes of that word coincide on the value of that term. Variable \( x \) corresponds to \( \text{self} \) in OCL-terms. It should be noted that properties (ii) and (iii) are not equivalent. Condition (ii) says that the terms corresponding to both words have the same value in both states. Condition (iii) says that functions corresponding to attributes act in the same way on term values. By induction on the word-length it can be proved that (iii) implies (ii). As an example, we consider the language consisting of words of the form \( \text{first}.\text{next}^n \) (see subsection 3.3). Two lists are indistinguishable, if terms \( \text{self}.\text{next}^n \) have the same values for both of them. As an example, we can consider networks of objects like in the class diagram on Figure 2, but without the assumption that there is at most one next element. Assume that in one state, a ‘relaxed list’ \( l \) consists of an anchor element \( a \) with follower \( e_1 \), \( e_1 \) has two followers \( e_{21} \) and \( e_{22} \), \( e_{21} \) has one follower \( e_{31} \), and \( e_{22} \) has one follower \( e_{32} \). Assume that in another state, the ‘relaxed list’ consists also of the anchor element \( a \) with follower \( e_1 \), \( e_1 \) has two followers \( e_{21} \) and \( e_{22} \), \( e_{21} \) has follower \( e_{31} \), but \( e_{22} \) has followers \( e_{31} \) and \( e_{32} \). In both states, the value of \( \text{first}.\text{next} \) at \( l \) equals \( \text{Set}\{e_{21}, e_{22}\} \) and the value of \( \text{first}.\text{next}.\text{next} \) equals \( \text{Set}\{e_{31}, e_{32}\} \). Thus condition (ii) is satisfied at \( l \). However condition (iii) is violated since \( e_{22} \) has different sets of followers in the above defined states.

We say that two states are locally indistinguishable in respect to a partial language \( L \), if they are indistinguishable in respect to the corresponding total language \( L \cap T^* \). The next lemma follows easily by induction on the length of words.

**Lemma 5.1.** Under assumptions of the definition above, property (ii) follows from property (iii). Moreover, if \( L \) contains a language of the form \( L_1 L_2 \), \( L_2 \) is prefix-closed and if \( o \in M[s \mapsto \sigma, x \mapsto o_p] \) (resp.) for a word \( w \in L_1 \), then \( \sigma \) and \( \sigma' \) are locally indistinguishable at \( o \) in respect to \( L_2 \).

The following theorem says that for every two states, a value of an OCL-term for a given argument is the same if the network of objects defined by the language corresponding to the term is the same in both states. Consequently, a network of objects defined by a term in a given state determines the value of the term in this state. In case of the list example, it implies that the term \( \text{self}.\text{elements} \) has the same values for two different states if they are locally indistinguishable for the above-mentioned language. In this theorem we consider terms which contain the restricted set of OCL primitives and no class attributes, as explained in subsection 4.2. For a state \( \sigma \), a class \( C \) and a model \( M \), \( C^M_\sigma \) denotes the set of objects of this class which exist in state \( \sigma \) (see subsection 5.1).
Theorem 5.1. Let \( t(x_1 : C_1, \ldots, x_n : C_n) \) be an OCL-term of the restricted syntax which is well typed in respect to a class model. Let \( \nu \) be a valuation of the form \( [s \mapsto \sigma, x_1 \mapsto o_1, \ldots, x_n \mapsto o_n] \) such that \( o_i \in C_{a_i}^{\mathbb{M}_f} \cap C_{a_i}^{\mathbb{M}_f} \), for \( i = 1, \ldots, n \); and let \( \nu' \) be a valuation which differs from \( \nu \) by mapping state \( s \) to \( \sigma' \) instead of \( \sigma \). Let \( \sigma \) and \( \sigma' \) be indistinguishable at \( o_i \) in respect to \( \mathcal{L}_{x_i}(t) \), for \( i = 1, \ldots, n \). Then \( t^{\mathbb{M}_f} \nu = t^{\mathbb{M}_f} \nu' \).

Proof:
Recall that a term \( t \) is defined for an argument \( \sigma = (o_1, \ldots, o_n) \) in the model \( \mathbb{M}_f \) and a state \( \sigma \) if, and only if, there is an unwinding \( u \) of \( t \) such that \( u \) is different from \( \bot \) in \( \mathbb{M}_f \uparrow \downarrow \) for the argument \( \sigma \) and the state \( \sigma \). We prove the following sublemma first:

Let the assumptions of this theorem be satisfied for \( \mathcal{P}_{\mathcal{L}_x}(t) \) instead of \( \mathcal{L}_{x_i}(t) \). Then the following conditions hold:

1. \( t(s, x_1, \ldots, x_n)^{M_f \uparrow \downarrow} \nu = t(s, x_1, \ldots, x_n)^{M_f \uparrow \downarrow} \nu' \) and \( t(s, x_1, \ldots, x_n)^{M_f \uparrow \downarrow} \nu \) is either equal \( \bot \) or it consists of objects and/or values of a basic OCL type.

2. If \( t \) defines a set of objects, then for every object \( o \in t^{M_f \uparrow \downarrow} \nu \), there exist a variable \( x_i \) and a total word \( \nu \in \mathcal{P}_{\mathcal{L}_x} \) such that \( o \in w^{M_f \uparrow \downarrow}(\sigma, \nu(x_i)) = w^{M_f \uparrow \downarrow}(\sigma', \nu(x_i)) \).

We prove this sublemma by structural induction. If \( t \) is atomic, then these properties hold trivially. Assume that they hold for all terms of height smaller than the height of \( t \). In case of state independent operations different from collect, conditions (1) and (2) follow directly from the inductive assumption.

Assume that \( t \) has the form \( (s, r)a \) for an object-property \( a \), a term \( r \) and a state variable \( s \). From the fact that \( \mathcal{P}_{\mathcal{L}_x}(r) \subseteq \mathcal{P}_{\mathcal{L}_x}(t) \) it follows that \( \sigma \) and \( \sigma' \) are locally indistinguishable in respect to \( \mathcal{P}_{\mathcal{L}_x}(r) \), for every variable’s index \( i \). From the fact that \( t \) is well typed and from the inductive assumption it follows that \( r^{M_f \uparrow \downarrow} \nu = r^{M_f \uparrow \downarrow} \nu' \) and that \( r^{M_f \uparrow \downarrow} \nu \) is either equal \( \bot \) or its value is a set of objects and/or values of a basic OCL type. Associations and attributes are defined only on objects, not on values of a basic type (see subsection 4.2); therefore the value of \( r \) is either a set of objects or \( \bot \). If \( r^{M_f \uparrow \downarrow} \nu = r^{M_f \uparrow \downarrow} \nu' = \bot \), then \( t^{M_f \uparrow \downarrow} \nu = t^{M_f \uparrow \downarrow} \nu' = \bot \). If not, then \( r^{M_f \uparrow \downarrow} \nu \) is a set of objects, possibly empty, on which the object-property \( a \) is defined.

From the inductive assumption it follows that for every object \( o \in r^{M_f \uparrow \downarrow} \nu \) there is a variable \( x_i \) and a total word \( w_1 \in \mathcal{P}_{\mathcal{L}_x}(r) \) such that \( o \in w_1^{M_f \uparrow \downarrow}(\sigma, \nu(x_i)) \). Clearly \( w_1a \in \mathcal{P}_{\mathcal{L}_x}(t) \).

From the local equivalence assumption it follows that \( (\sigma, o)a^{M_f \uparrow \downarrow} = (\sigma', o)a^{M_f \uparrow \downarrow} \), \( t^{M_f \uparrow \downarrow} \nu = t^{M_f \uparrow \downarrow} \nu' \) and \( t^{M_f \uparrow \downarrow} \nu \) is composed of objects or values of a basic type. If \( o \in t^{M_f \uparrow \downarrow} \nu \), then there exists an object \( o_1 \in t^{M_f \uparrow \downarrow} \nu \) such that \( (\sigma, o_1)a^{M_f \uparrow \downarrow} \) includes \( o \). From the inductive assumption it follows that there is a variable \( x_i \) of \( r \) and a word \( w_1 \in \mathcal{P}_{\mathcal{L}_x}(r) \) such that \( o_1 \in w_1^{M_f \uparrow \downarrow}(\sigma, \nu(x_i)) \). We can define \( w \) to be equal \( w_1a \).

If \( t \) has the form \( f(t_1, \ldots, t_k) \) for an inductively defined symbol \( f \), then \( f(t_1, \ldots, t_k) \) is constantly equal to \( \bot \) in \( M_f \uparrow \downarrow \).

If \( t \) has the form \( t_1 \rightarrow \text{collect}(y \mid t_2) \), then from the inductive assumption it follows that the states are locally indistinguishable in respect to \( \mathcal{P}_{\mathcal{L}_x}(t_1) \) at \( o_i \); consequently \( t_1^{M_f \uparrow \downarrow} \nu = t_1^{M_f \uparrow \downarrow} \nu' \). Similarly for \( t_2 \) and variable \( x_i \). We show now that for every \( o_\rho \in \text{collect}(t_1^{M_f \uparrow \downarrow} \nu, \sigma \text{ and } \sigma' \text{ are locally} \)
indistinguishability at $o_p$ in respect to $\mathcal{P} \mathcal{L}^O_y(t_2)$. From the inductive assumption it follows that there is an object $x$ such that $\mathcal{P} \mathcal{L}^O_x(t_1) \subseteq \mathcal{P} \mathcal{L}^O_x(t_2)$ and since the states are locally indistinguishable in respect to $\mathcal{P} \mathcal{L}^C_x(t)$, lemma 5.1 and the fact that closed pre-languages are prefix-closed imply local indistinguishability at $o_p$ in respect to $\mathcal{P} \mathcal{L}^C_y(t_2)$. Consequently, from the inductive assumption and the sublemma assumption it follows that $t_2^{M \mapsto \perp} v[y \mapsto o_p] = t_2^{M \mapsto \perp} v'[y \mapsto o_p]$ holds for every $o_p \in t_1^{M \mapsto \perp} v$. Consequently, $t_1 \rightarrow \text{collect}(y | t_2)^{M \mapsto \perp} v = t_1 \rightarrow \text{collect}(y | t_2)^{M \mapsto \perp} v'$. Let $o$ belong to $t_2^{M \mapsto \perp} v$. We prove now the existence of a word leading to $o$. From the definition of the collect-operation it follows that there is an object $o_p = t_1^{M \mapsto \perp} v$ such that $o$ belongs to $t_2^{M \mapsto \perp} v[y \mapsto o_p]$ (see subsection 5.1). From the inductive assumption it follows that there exist a variable $z$ of $t_2$ and a word $w_2 \in \mathcal{P} \mathcal{L}^C_x(t_2)$ leading from $(z)v[y \mapsto o_p]$ to $o$. If $z$ and $y$ are different, then the existence follows from the inductive assumption. If these variables are equal, then there exist words $w_1 \in \mathcal{P} \mathcal{L}^O_x(t_1)$ and $w_2 \in \mathcal{P} \mathcal{L}^C_y(t_2)$ such that $w_1w_2$ leads to $o$.

Let $t^{M}v$ be different from $\perp$. By definition, there is an unwinding $u$ of $t$ such that $t^{M}v = u^{M \mapsto \perp}v$. If $L_x(u) \subseteq L_x(t)$, for every variable $x$ of $t$, then the sublemma implies that $u^{M}v = u^{M \mapsto \perp}v'$ and consequently $t^{M}v = u^{M \mapsto \perp}v = u^{M}v = t^{M}v'$. Points (2) and (1) of corollary (4.1) imply that if $w \in L_x(u)$, then there exists an unwinding $v$ of $u$ such that $w^C \in \mathcal{P} \mathcal{L}^C_x(v)^C$. $w^C$ is obtained from $w$ by adding the superscript $C$ to the last terminal (cf. subsection 4.4). Term $v$ is an unwinding of $t$, thus $s_t \Rightarrow^* w^C$, and $w \in L_x(t)$ since we can apply a rule eliminating the superscript.

\[ \square \]

For an OCL-term $t$, $\text{Prop}(t)$ denotes the set of all object-properties, i.e., object attributes and association-ends, occurring in $t$; $\text{Fun}(t)$ denotes the set containing all recursively defined function symbols $f_j$ occurring in $t$. We define the set of relevant properties corresponding to $t$ as the smallest set of symbols defined by the equation $\text{RelProp}(t) = \text{Prop}(t) \cup \bigcup_{f \in \text{Fun}(t)} \text{RelProp}(f_j)$. This set includes all properties occurring in $t$ or in definitions of recursively defined function symbols occurring in $t$ or in the definitions of symbols occurring in the definitions and so on. In case of the list example, $\text{RelProp(\text{elements})} = \{ \text{first, next} \}$.

Let $t$ be an OCL-term, let $x$ be a variable of class $C$, let $L_x(t)$ be the corresponding language and let $o_p$ be an object of class $C$, which exists in state $\sigma$ (i.e., $o_p \in C^\sigma$). By $(\sigma, o_p)L_x(t)^{M}$ we denote the set of all values reachable from $o_p$ in state $\sigma$ via a path marked with a word $w \in L_x(t)$; i.e., $(\sigma, o_p)L_x(t)^{M} = \bigcup_{w \in L_x(t)} (w^{M} | s \rightarrow \sigma, x \rightarrow o)$, where $w(s : \text{State}, x : C)$ is a term corresponding to the word $w$.

Let $\sigma$ and $\sigma'$ be states. Let $a$ be an object-property with arguments of class $C$. Let $t(s : \text{State}, x_1 : B_1, \ldots, x_n : B_n) : \text{Collection}(C)$ be a formalization of an OCL-term. Let $v$ be a valuation such that $v(s) = \sigma$ and $v(x_i) \in B_i^\sigma$, for $i = 1, \ldots, n$. We say that $t$ defines the scope of change of $a$, if, and only if, the following holds (cf. [26, 27]): $\forall \sigma, o \in C^\sigma \cap C^\sigma' \land (\sigma, o) a \neq (\sigma', o) a \implies o \in t^{M}v$.

The definitions above are valid for models $M_f$ and $M^{\perp}$ as well. The scope of change is defined in the pre-state, therefore we assume that $v$ maps variables $x_i$ on objects existing in the pre-state.

As mentioned in subsection 4.2, an OCL-term containing class attributes can be obtained by substitution from another term which does not contain such attributes. Let $t(x_1 : C_1, \ldots, x_n : C_n)$:
Let \( \text{self}_1 : \text{C}_1, \ldots, \text{self}_n : \text{C}_n \) be an OCL-term containing class attributes \( D_1 \cdot d_1 \) of type \( \text{T}_1 \), for \( i = 1, \ldots, m \). We can present \( t \) in the form \( t_0 [D_1 \cdot d_1 / z_1, \ldots, D_m \cdot d_m / z_m] \) where \( t_0 \) does not contain class attributes. In case of class attributes, we consider paths starting with a class attribute. Therefore, we consider terms of the form \( w[C.c / x] \), which are obtained from \( w \) by substituting the constant \( C.c / x \) for the free variable \( x \). Note that the unwindings of \( t \) are exactly matched by the unwindings of \( t_0 \); i.e., if an unwinding is done at a position in term \( t_0 \) and results in term \( u_0 \), then the same unwinding can be performed in term \( t \) and results in term \( u_0[D_1 \cdot d_1 / z_1, \ldots, D_m \cdot d_m / z_m] \), and vice versa. This follows directly from the fact that recursive definitions can be applied to function symbols only, but not to class attributes.

The next corollary follows from theorem 5.1. It says that preservation of an invariant depends on the preservation of object networks corresponding to the language of the invariant. After a state change, if the network of properties traversed during invariant evaluation does not crosscut scopes of change of relevant properties, then the invariant validity is preserved. Condition (1) says that the invariant is evaluated at the same points in the consecutive states and that the terms defining the scope of change of object-properties are evaluated at objects existing in the pre-state (cf. [26]). Condition (2) says that the paths navigated from points where the invariant is evaluated do not crosscut the scope of change of object-properties. Condition (3) says that the class attributes occurring in the invariant do not change. Condition (4) is similar to condition (2) and says that the paths navigated from class attributes do not crosscut the scope of change of object-properties. It should be noted that \( \nu' \) does not need to map variables of the term defining the scope of change since this term is evaluated in the pre-state. Recall that we treat formulas, in particular invariants, as boolean valued terms.

**Corollary 5.1.** Let \( \text{I} (\text{self}_1 : \text{C}_1, \ldots, \text{self}_n : \text{C}_n) \) be an OCL-term of the form \( \text{I}_0 [D_1 \cdot d_1 / z_1, \ldots, D_m \cdot d_m / z_m] \) where \( \text{I}_0 \) is of the restricted syntax. Let \( \text{RelProp}(\text{I}_0) \) have the form \( \{a_1, \ldots, a_k\} \). For \( j = 1, \ldots, k \), let \( t_j (x_1 : \text{B}_1, \ldots, x_r : \text{B}_r) \) define the scope of change of property \( a_j \). Let \( \sigma \) and \( \sigma' \) be two states. Let \( \nu = [s \mapsto \sigma, \text{self}_1 \mapsto o_{p_1}, \ldots, \text{self}_n \mapsto o_{p_n}, x_1 \mapsto o_1, \ldots, x_r \mapsto o_r] \) and \( \nu' = [s \mapsto \sigma', \text{self}_1 \mapsto o_{p_1}, \ldots, \text{self}_n \mapsto o_{p_n}] \) be a valuations. If the following conditions are satisfied:

1. \( o_{p_i} \in C_{i\sigma}^{\nu\sigma} \cap C_{i\sigma'}^{\nu\sigma'}, \) for \( i = 1, \ldots, n \), and \( o_j \in B_{j\sigma}^{\nu\sigma} \), for \( j = 1, \ldots, r \)
2. \( (\sigma, o_{p_i})(\text{L}_{\text{self}_i}(\text{I}_0)/a_j)^{\nu\sigma} \cap t_j^{\nu\sigma} \nu = \emptyset, \) for \( i = 1, \ldots, n \) and \( j = 1, \ldots, k \)
3. \( (\sigma)D_{i_i}d_i^{\nu\sigma} = (\sigma')D_{i_i}d_i^{\nu\sigma}, \) for \( i = 1, \ldots, m \)
4. \( (\sigma, o_i)(\text{L}_{z_i}(\text{I}_0)/a_j)^{\nu\sigma} \cap t_j^{\nu\sigma} \nu = \emptyset, \) for \( o_i \in D_i.d_i^{\nu\sigma}, i = 1, \ldots, m \) and \( j = 1, \ldots, k \)

Then \( I^{\nu\sigma} \nu = I^{\nu\sigma'} \nu', \) where \( I \) is the formalization of the OCL-term \( \text{I} \).

**Proof:**
Let the assumption of this corollary hold. From theorem 5.1 it follows that it is enough to prove that the states are locally indistinguishable at \( o_{p_i} \), for \( i = 1, \ldots, n \), and at \( o_j \), for \( j = 1, \ldots, m \). The OCL-terms corresponding to total words in \( \text{L}_{\text{self}_i}(\text{I}_0)/a_j \) and \( \text{L}_{z_j}(\text{I}_0)/a_j \) are well typed (see lemma 4.4 in subsection 4.5). The third property of definition 5.1 follows from points (2) and (4). The second property follows from the third one according to lemma 5.1. \( \square \)
5.3. Expressing Reversed Grammars in OCL

In this subsection we show how to define OCL-terms allowing navigation from the scope of change of a property to the roots of the affected invariant networks and how to use those terms to explicitly define OCL post-conditions corresponding to invariants. As pointed by Barnett et al. [8], it is not enough to check invariants validity only as a part of a method’s post-condition. An invariant has to be checked also before public method calls. However, we define the backward navigation terms as part of post-conditions, to give them some context. In section 7, we discuss this problem in a general setting.

Let \( I(\text{self}_1 : C_1, \ldots, \text{self}_n : C_n) \) be an invariant of the form \( I_0[D_1.d_1/z_1, \ldots, D_n.d_n/z_n] \), where \( I_0 \) does not contain class attributes, and let operation \( \text{op} \) be defined by the following constraint:

\[
\begin{align*}
\text{context } D::\text{op}(x_1 : B_1, \ldots, x_r : B_r) \\
\text{pre} & : \text{Pre} \\
\text{post} & : \text{Post} \\
\text{modifies} & : t_1::a_1, \ldots, t_k::a_k, E_1.e_1, \ldots, E_1.e_1
\end{align*}
\]

The modifies-clause says that method \( \text{op} \) can only change object attributes \( a_j \), for \( j = 1, \ldots, k \), and class attributes \( E_1.e_1 \), for \( i = 1, \ldots, l \), which may or may not be of the form \( D_1.d_i \). The scope of change of attribute \( a_j \) is defined by term \( t_j \) (cf. [26]).

In this subsection, we use the grammar obtained from \( (T, Nt/a, s_{xt/a}, \gamma) \) by removing all nonterminals which cannot be reduced to total words (see the end of subsection 4.4). \( G_{\text{self}_1}(I_0)/a_j \) is a grammar allowing navigation from a root of the invariant to objects having the property \( a_j \) (see the previous subsection). Observe that if \( a_j \) is an object-property with arguments of class \( A_j \), then for every word \( w \in L(G_{\text{self}_1}(I_0)/a_j) \) the word allows us to navigate from an object of class \( C_1 \) to objects of class \( A_j \) and consequently the corresponding formal term \( w \) is of type \( A_j \), or \( \text{Collection}(A_j) \) (see subsection 4.5). Thus, for the start symbol of this grammar, we can define a function which maps \( C_1 \) to \( \text{Collection}(A_j) \). Similarly for a recursively defined function symbol \( f(y_1, \ldots, y_j) \) of type \( T \) and a variable \( y_i \) of type \( A \), we can define a function which maps objects of class \( A \) to \( \text{Collection}(T) \); this function corresponds to term \( (x : A).f_{y_1} : \text{Collection}(T) \) (see subsection 4.5). We associate this function with the corresponding nonterminal \( f_{y_1} \). With a nonterminal \( f_{y_i}^C \) and a variable \( y_i \) of type \( C \) we can associate a function with arguments of type \( C \) and values of type \( \text{Collection}(A_j) \) (see subsection 4.5). Below, \( f_{y_j}^C \) denotes a term of the form \( f_{y_j} \) or of the form \( f_{y_j}^C \). In subsection 4.5, it is proved that in fact we can eliminate nonterminals \( f_{y_i} \) and \( f_{y_i}^C \) if \( y_i \) is not of a class type, and nonterminal \( f_{y_i} \) if \( y_i \) is not of a class type since those nonterminals cannot occur in partial words reducible to total ones.

Now we show how to define OCL-terms allowing navigation from the scope of change of property \( a_j \) to the roots of the affected invariant’s networks. For every context free grammar \( G \) and the corresponding language \( L \), we can define language \( \overline{L} = \{a_n \ldots a_1 | a_1 \ldots a_n \in L \} \) which contains the reversed words. The corresponding reversed grammar \( \overline{G} \) has exactly the same terminals and nonterminals as \( G \), but every rule \( Y \rightarrow A_1 \ldots A_n \) of \( G \) is replaced by the rule \( Y \rightarrow A_n \ldots A_1 \). Let \( (G_{\text{self}_1}(I_0)/a_j)^{rev} \) be the reversed grammar corresponding to \( G_{\text{self}_1}(I_0)/a_j \). Note that we cannot simply apply words of this grammar to navigate from the scope of change of \( a_j \) to the roots of the
affected invariant’s networks since it is not possible to navigate to the opposite end of an association using the same association-end’s name. We have to use the opposite association-end. Thus, for every association-end b allowing a navigation from a classes A to a class B, we define the inverse association-end \( b^{rev} \) as follows:

\[
\text{context } B \text{ def : } \\
self.b^{rev} : \text{ Collection}(A) = \\
A.\text{allInstances()}\rightarrow\text{collect}(o | \: o.b\rightarrow\text{includes}(self))
\]

In case of a nonterminal Y corresponding to a start symbol of the form \( s_{self, I_0/a_j} \) or to a recursively defined symbol (see subsection 4.4) we introduce an equally named OCL function \( Y^{rev} \). I.e. the start symbol \( s_{self, I_0/a_j} \) corresponds to the OCL function \( s_{self, I_0/a_j}^{rev} \) and a nonterminal of the form \( f_{i,y_k}^{\Box} \) corresponds to the OCL-term \( f_{i,y_k}^{\Box} \). If \( self_i \) is of type \( C_i \) and \( a_j \) has argument of type \( A_j \), then \( s_{self_i, I_0/a_j}^{rev} \) has arguments of type \( A_j \) and values of type \( \text{Collection}(C_i) \); similarly for \( f_{i,y_k}^{\Box} \).

In order to navigate in an opposite direction via a path marked by association ends we may need to downcast the resulting objects. For example, if \( b \) allows us to navigate from class \( A \) to class \( B \), \( B \) is a subclass of \( D \) and \( e \) allows navigation from \( D \) to \( E \), then we cannot just compose \( e^{rev} \) and \( b^{rev} \) to navigate from \( E \) to \( A \). We need to select objects of class \( B \) among objects of class \( D \) and to downcast them in order to navigate via \( e^{rev} \) and \( b^{rev} \). In the following, terms of the form

\[
x\rightarrow\text{select}(o \mid o.\text{oclIsKindOf}(C))\rightarrow\text{oclAsType}(C)
\]

are abbreviated as \( x\rightarrow\text{selectAs}(C) \). Let the nonterminal Y correspond to a function symbol with an argument of type \( C \) and values of type \( \text{Collection}(D) \) (see above). For reduction rules of the form

\[
Y \rightarrow W_1 | \ldots | W_m
\]

of the reversed grammar \( (G_{self, I_0/a})^{rev} \) we introduce function:

\[
\text{context } D \text{ def : } \\
Y^{rev} : \text{ Collection}(C) = \self.W_1^{rev}\rightarrow\text{union}(\ldots \\
\rightarrow\text{union}(\self.W_m^{rev})\ldots)\rightarrow\text{selectAs}(C),
\]

where \( W_i^{rev} \) is an OCL-term obtained from partial word \( W_i \) by replacing every terminal \( b \) corresponding to an association-end between classes \( A \) and \( B \) by OCL-term \( \rightarrow\text{selectAs}(B).b^{rev} \) and every nonterminal \( f_{i,y_k}^{\Box} \) corresponding to variable \( y_k \) of type \( A \) and function \( f \) with values of type \( B \) \( (\text{Collection}(B), \text{resp.}) \) by term \( \rightarrow\text{selectAs}(B).f_{i,y_k}^{\Box} \). In particular the start symbol \( s_{self_i, I_0/a} \) is replaced with the OCL function symbol \( \rightarrow\text{selectAs}(C).s_{self_i, I_0/a}^{rev} \).

Note that we have to downcast the arguments of the reversed operation \( f_{i,y_k}^{\Box} \) as we do it in case of association-ends. Terms of the form \( Y^{rev} \) are well typed since they are obtained by reversing a path and downcasting to proper classes so that the associations can be composed. In subsection 4.5, we showed that the OCL-terms corresponding to reversed partial words are type correct.

The lemma below follows from basic properties of relational composition and corollary 5.1. It says that the above-defined procedure can be used to navigate backwards to the roots of modified networks. Such a root can be either an object or a class attribute. In the second case, either a path leading from the class attribute or the class attribute itself is modified.

**Lemma 5.2.** Let term I, valuations \( \nu \) and \( \nu' \), states \( \sigma \) and \( \sigma' \) and terms \( t_i \) satisfy the assumption of corollary 5.1. Moreover, let its condition (1) be satisfied. If \( I^{\nu'} \neq I^{\nu} \), then there exists an object-property \( a_j \) and variable \( self_i \) such that \( \nu(self_i) \in ((s, t_j)s^{\nu}_{self_i, I_0/a_j})^{\nu} \), for some \( i =
1, ..., n, or there exists an object attribute $D_i \cdot d_i$ such that $D_i \cdot d_i(s)^{Mf} \cap ((s, t_j)s_{z_i}^{rev}I_0\backslash a_j)^{Mf} \neq \emptyset$, for an $i = 1, ..., m$, or $D_i \cdot d_i(s)^{Mf} \neq D_i \cdot d_i(s)^{Mf}'$ for an $i = 1, ..., m$.

**Proof:**
We present here a proof sketch. If the values of $I$ are different for valuations $v$ and $v'$, then condition (2), (3) or (4) of corollary 5.1 is violated. For every binary relation $R$ and every set $A$, if $R(A) \subseteq B$, then $A \subseteq R^{-1}(B)$. The composition of binary relations is a binary relation. Consequently, if a word $w$ marks a path from $o_1$ to $o_2$, then the reversed word allows to navigate from $o_2$ to $o_1$ after performing the downcasting. If $I$ has different values for $v$ and $v'$, then a scope of change of a property occurring in $I$ crosses a path leading from an actual parameter or a class attribute of $I$ to value of $I$. Thus, knowing the scope of change, we can use the reversed grammar to navigate to the roots of modified networks. The recursively defined OCL functions $s_{z_i}^{rev}_I o_1/a_j$ and $s_{z_i}^{rev}_I o_0/a_j$ determine all objects backward navigable from the scopes of change using paths determined by the reversed grammars. Therefore, those functions identify the roots. ☐

Invariant evaluation after method execution is time consuming. Lemma 5.2 allows us to restrict the scope of the evaluation. We consider now invariants of the form described at the beginning of this section. From the corollary 5.1 and from statement 5.2 it follows that to prove that such an invariant is valid it is enough to prove the following:

1. The scopes of change of properties do not crosscut invariant evaluation paths leading from class attributes
2. The invariant holds for all pre-existing objects which are roots of networks crosscutting the scopes of properties’ change
3. The invariant holds for all objects created during a method’s execution

If condition (1) is not satisfied, i.e., a path labelled with a word from $L_{z_i}(I_0)$ is modified, then the invariant has to be evaluated for all objects. In the other case the invariant has to be evaluated for all objects being roots of modified networks, i.e., if there is a modified path labelled with a word from $L_{z_i}(I_0)$, and for the newly created ones. An invariant may include several variables meaning that it has to be evaluated for tuples of objects. Actually most invariants contain only one variable. For example, in the book [43] all invariants contain only one free variable. If objects in a tuple are not new, the corresponding networks remain unchanged and in addition no path leading from class attributes is modified, then it is not necessary to evaluate the invariant for the tuple. In case when no paths leading from class attributes are modified, it is necessary to evaluate the invariant for tuples where at least one object is newly created or at least one object is a root of modified network. In particular, if the properties occurring in the invariant do not occur in the modifies-clause, then it is enough to evaluate the invariant only for newly created objects since the paths leading from the old objects and class attributes cannot be modified.
6. Examples Continued

In this section we apply our method to examples presented in section 3. In particular, we demonstrate how to extract context free languages from recursive definitions and OCL-terms and how to define post-conditions corresponding to invariants.

6.1. Airline Example Continued

In the airline example (see subsection 3.2), there are no recursive definitions and the application of our method is rather straightforward. However, this example shows how to identify roots of invariant networks modified by a method and how to deal with class attributes.

The invariant \( \text{tax}_\text{inv} \) includes one variable \( \text{self} \) and quantifier \( \text{forAll} \). Let \( \text{Eqv} \) denote the term: \( p.\text{highEarnings} \iff p.\text{role}.\text{salary}\rightarrow\text{sum}() \geq 2000 \). The general quantifier can be expressed using \( \text{collect} \):
\[
\text{not}(\text{self}.\text{people}\rightarrow\text{collect}(p \mid \text{Eqv})\rightarrow\text{includes}(\text{false}))
\]
Since the evaluation of \( \text{not} \) does not change the language and since the closed pre-language corresponding to the second argument of \( \text{includes}(\text{false}) \) is empty, the language \( \text{Pre}_\text{self}(\text{tax}_\text{inv}) \) corresponding to the invariant has the following form:
\[
\text{Pre}_\text{self}(\text{self}.\text{people})\text{Pre}_p(\text{Eqv}) \cup \text{Pre}_\text{self}(\text{self}.\text{people}) \cup \text{Pre}_\text{self}(\text{Eqv}),
\]
where:
\[
\text{Pre}_p(\text{Eqv}) = \text{Pre}_p(\text{p.\text{highEarnings}}) \cup \text{Pre}_p(\text{p.\text{role}.\text{salary}}) = \\
\{ \epsilon, \text{highEarnings}, \text{role}, \text{role salary} \}
\]
Set \( \text{Pre}_\text{self}(\text{Eqv}) \) is empty; consequently, the closed pre-language corresponding to the invariant has the following form: \( \{ \epsilon, \text{people}, \text{people role}, \text{people role salary}, \text{people highEarnings} \} \). Note that this language is prefix-closed. Division by salary results in the singleton \( \{ \text{people role} \} \). Similarly, the invariant \( \text{airline}_\text{inv} \) corresponds to the language \( \{ \epsilon, \text{pilots}, \text{pilots salary} \} \). Division by salary results in \( \{ \text{pilots} \} \).

The method \( \text{raise} \) changes only the attribute salary of the implicit actual parameter and may create some new objects. Its execution may result in making both invariants invalid due to modification of the attribute salary. We can trace back the changes by dividing corresponding languages by salary and reversing them. The reversed languages uncover all roots of invariant networks which may be invalidated by the execution. The creation of new objects of classes \( \text{Role} \) and \( \text{Person} \) does not violate those two invariants. Let \( C->\text{selectNew}() \) be an abbreviation for \( C.\text{allInstances}()->\text{select}(o \mid o.\text{isNew}()) \). The specification of \( \text{raise} \) can be therefore presented in the following way:

```plaintext
context Role::raise(s : Real)
post : self.salary = self.salary@pre + s and
\[
\text{Set}\{\text{self}\}->\text{selectAs(Pilot)}->\text{pilots}^{\text{rev}}->\text{selectAs(Airline)}
\rightarrow\text{union}(\text{Airline.\text{selectNew}()})\rightarrow\text{forAll}(\text{al : Airline | al.\text{airline}_\text{inv}}) \text{ and }
\text{Set}\{\text{self}\}->\text{selectAs(Role)}->\text{role}^{\text{rev}}->\text{selectAs(Person).\text{people}^{\text{rev}}}
\rightarrow\text{selectAs(Taxman)}->\text{union}(\text{Taxman->\text{selectNew}()})
\rightarrow\text{forAll}(\text{ta : Taxman | ta.\text{tax}_\text{inv}})
\]
modifies : self.salary
```
This example explains why selection and downcasting is necessary when opposite association ends are navigated through. We cannot navigate from class `Role` back to class `Airline` without selecting objects of class `Pilot` and downcasting them. On the other hand, some of the select and downcast terms are redundant. The post-condition is actually not realizable for some values of the parameter `s` since in some cases a pilot’s salary may be decreased beyond the specified limit, so the pre-condition has to be strengthened.

Let `ca` be a class attribute occurring in the set of relevant properties corresponding to an invariant. If `ca` is changed, then the invariant must be evaluated for all combinations of objects. This is due to the fact that class attributes resemble object attributes having the same value for all objects. The change of `ca` is like the simultaneous change of all objects. For example, consider the following invariant:

```plaintext
context Taxman inv threshold_inv :
    self.people->forAll(p : Person | p.highEarnings iff p.role.salary->sum() >= Taxman.tr)
```

The difference between this invariant and `tax_inv` is that the total salary is compared not to a fixed real number, but to the value of a class attribute. If this attribute is changed, then the invariant has to be re-evaluated for all objects of class `Taxman`. The following constraint specifies the method `setTr`:

```plaintext
context Taxman::setTr(x : Real)
post : tr = x
modifies : Taxman.tr
```

In case of `setTr` and the invariant `threshold_inv`, our extraction method does not allow to reduce the search space. However, as mentioned above, it allows us to identify cases when the re-evaluation of the invariant is not necessary.

### 6.2. List Example Continued

In this subsection we reconsider the list example (see subsections 3.3, 4.2 and 4.4). It uses recursion to define the set of list elements. There is only one language corresponding to the invariant `no_loops` since it includes only one variable and it does not include class attributes. It has the form `ε | first next*`. It should be noted that this language is regular. Division by `next` results in language `first next*`.

In the definition of OCL-terms for backward navigation we skip the downcasting part since it is not necessary here. We make also the assumption that for any type `T`, it is subtype of `Collection(T)` (see subsection 4.1).

```plaintext
context Element
def : s_rev no_loops/next = elements_rev ->union(next_rev.elements_rev)
    ->union(succOf_acc_rev.first_rev)
def : succOf_self_rev : Collection(Element) = x->union(next_rev)->union(succOf_self_rev.next_rev)
```
Function \texttt{succOf}^C_{AcC} can be defined in a similar way.

The language corresponding to the invariant \texttt{inv.total} has the form: $\epsilon \mid \texttt{total} \mid \texttt{first next}^* \mid \texttt{first next}^* x$. Since \texttt{total} is an attribute of anchor objects, its change in one \texttt{List} object does not require checking the invariant for any other object of this class. However, if we manipulate the association-end \texttt{next}, then we have to check if the invariant holds for all list anchors which are backward-navigable from the objects for which association-end \texttt{next} was modified.

The case of the \texttt{no_sharing} invariant is different from the preceding ones in that it contains two variables. Brute-force validation of this invariant requires checking all pairs of anchor objects of class \texttt{List}. If the number of existing lists is high, then this process may be very time consuming. The method \texttt{award} cannot violate this invariant unless it creates some new lists; in this case it is enough to check pairs consisting of a newly created list and a pre-existing one. Only the execution of \texttt{insert} can violate this invariant. In this case, our method allows us to restrict invariant validation to pairs consisting of an anchor which is backward navigable for the object newly linked to the inserted object and of an arbitrary anchor. Thus, the product search is reduced, in a sense, to a one dimensional space.

### 7. Implementation and Applications

In this section we discuss a possible implementation of presented concepts. However, it should be noted that a profound treatment of the implementation issue goes far beyond the scope of this paper.

As mentioned in subsection 5.3, it is not enough to treat invariants as an implicit part of pre- and post-conditions, but they must be handled explicitly. Our method allows us to extract necessary conditions from invariants and modifies-clause, and to combine them with post-conditions. The situation is a bit more complicated in case of pre-conditions since it is not possible to express in OCL what was changed since the last public (helper, resp.) method was called or terminated.

In our description we refer to observable states, i.e., states when a public or non-helper method is called or terminates. The definition of observable state can be more general and include the state before and after any method call and/or property modification. Proposed invariant checking algorithm is neutral in respect to the definition of observable states.

We assume a number of invariants \texttt{self.I}_C, with one free variable \texttt{self} of a class \texttt{C}. We assume that those invariants have the form $I_0[D_1.d_1/z_1, \ldots, D_m.d_m/z_m]$ (see subsection 5.2). For every property \texttt{a} of class \texttt{A} we define class attribute \texttt{A.modif}_\texttt{a} of type \texttt{Collection(A)} to store objects of this class for which \texttt{a} is modified. Similarly for every class attribute \texttt{D.d}, we introduce new class attribute \texttt{D.modif}_\texttt{d} of the boolean type to record modification of \texttt{D.d}. Moreover, for every class \texttt{A}, we define class attribute \texttt{A.new}_\texttt{A} of type \texttt{Collection(A)} to store newly created objects of this class. After initializing all classes the invariants are checked, above defined collections of objects are initialized as empty sets, and class attributes \texttt{D.modif}_\texttt{d} are set to false.

1. Whenever a property \texttt{a} is modified, the corresponding object is stored in the collection \texttt{modif}_\texttt{a}

2. Whenever a class attribute \texttt{D.d} is modified, the corresponding attribute \texttt{D.modif}_\texttt{d} is set to true
3. Whenever a constructor of a class \( A \) is executed, the newly created object is stored in \( A.\text{new}_A \).

4. Whenever an observable state is entered or exited, the following is done: If a class attribute \( D.\text{modif}_d \) has value true and if an invariant \( I_C \) contains \( D.d \), then \( I_C \) is evaluated for all objects of class \( C \). If \( A.\text{modif}_a \) is not empty and object-property \( a \) occurs in a word starting with a class attribute \( D_j.d_j \), i.e., \( wa \in L_{z_j}(I_C) \), then \( I_C \) is evaluated for all objects of class \( C \); if there is no such a word then term \( s_{self, I_C/a}^{rev} \) is evaluated for every element of \( A.\text{modif}_a \). Moreover, invariant \( I_C \) is evaluated for all elements of \( \text{new}_C \).

5. After checking all invariants, all sets of the form \( A.\text{modif}_a \) and \( \text{new}_A \) are emptied. Similarly, all attributes \( D.\text{modif}_d \) are set to false.

This algorithm can be implemented using AspectJ in the way proposed by Van Der Straeten et al. [41]. In this language every method call can be caught by a so-called pointcut and then an action can be performed by a so-called advice. If an attribute is modified, then this modification can be registered as well, the corresponding object identified and stored in \( \text{modif}_a \). Similarly, constructor calls can be registered and the newly created objects identified.

This algorithm can be used at the runtime. There is of course also the question of how to compute the formulas \( s_{self, I_C/a}^{rev} \). Computing these formulas at the runtime would slow down the execution immensely. However, context free languages extracted from invariants and the corresponding OCL-terms can be computed at the compilation time. Similarly for all properties \( a \), it is possible to identify at the compilation time all invariants \( I_C \) such that there is a word \( wa \in L_{z_j}(I_C) \) for some class attribute \( D_j.d_j \). It does not amount for a verifying compiler as proposed by Hoare [20], but puts some of the computing burden on the compiler.

It should be pointed out that our method applies also to the case when invariants contain the predefined feature \( C.\text{allInstances}() \) (cf. [38]). It defines all instances of class \( C \) existing at the moment when the formula including this feature is evaluated. In general, this feature can be treated as a class attribute. Invariants containing \( C.\text{allInstances}() \) have to be re-evaluated when a new object of class \( C \) is created or deleted.

The process of extracting context free languages may result in grammars containing nonterminals, which are not reducible to total words. There is a method allowing us to eliminate such nonterminals without changing the resulting language (cf. e.g. [23]). There are also various other methods for the optimization of the compilation process and the resulting grammars (cf. e.g. [1]).

In case of implementation, there are a number of other issues one has to address. There is the issue of realizability and efficiency. In our case, we turn all associations into bidirectional ones. Akehurst et al. [4] discussed how different kinds of associations can be implemented in Java. They pointed out that implementing unidirectional associations as bidirectional ones has a number of advantages, but the disadvantage is that it hinders compositionality. In our case however, making all associations bidirectional can be hidden within aspects.
8. Conclusion and Future Work

In this paper we showed how to extract languages corresponding to OCL-invariants. We demonstrated that this method can be used to restrict the search space in case of invariant validation during the runtime. We discussed also shortly a possible implementation of presented ideas. Our work complements the work done in case of JML and Spec# on using types and special architectures to ease validation of invariants.

The examples considered in this paper result in regular languages; however, in general it would be possible to define data-types corresponding to context free languages. Algorithms for checking properties of regular languages are more efficient than general algorithms concerning context free languages. Investigating the complexity classes of invariants was not the goal of this paper, but in general results concerning complexity estimates of various logical formulas can be applied to estimate time needed to validate invariants. Presented results concern only associations with two ends. However, in general, associations can have multiple ends. In the future we are going to investigate these issues. We are also going to implement presented concepts.

References


