A Hoare Calculus for Verifying Java Realizations of OCL-Constrained Design Models

Bernhard Reus[1] and Martin Wirsing[2] and Rolf Hennicker[2]

[1] School of Cognitive and Computing Sciences, University of Sussex at Brighton
bernhard@cogs.susx.ac.uk, Fax +44 1273 671 320
[2] Institut für Informatik, Ludwig-Maximilians-Universität München
[hennicke|wirsing]@informatik.uni-muenchen.de

Abstract. The Object Constraint Language OCL offers a formal notation for constraining the modelling elements occurring in UML diagrams. In this paper we apply OCL for developing Java realizations of UML design models and introduce a new Hoare-Calculus for Java classes which uses OCL as assertion language. The Hoare rules are as usual for while programs, blocks and (possibly recursive) method calls. Update of instance variables is handled by an explicit substitution operator which also takes care of aliasing. For verifying a Java subsystem w.r.t. a design subsystem specified using OCL constraints we define an appropriate realization relation and illustrate our approach by an example.

1 Introduction

Program verification is a dream which has not yet been realized in practical software development. With UML [17] the possibilities for achieving this dream have improved: UML allows one to express semantic constraints using OCL and offers notations such as the "realizes" relation for expressing correctness relationships between different diagrams on different levels of abstraction. The object constraint language OCL [23] offers a formal notation to constrain the interpretation of modelling elements occurring in UML diagrams. OCL is systematically used for rigorous software development in the Catalysis Approach [11]. The OCL notation is particularly suited to constrain class diagrams since OCL expressions allow one to navigate along associations and to describe conditions for object attributes in invariants and pre- and post-conditions of the operations. The "realizes" relationship asserts that classes (written in a programming language) "realize" the requirements formulated in a more abstract class diagram with constraints. It allows the programmer to express the correctness of its implementations w.r.t. UML designs. However, to our knowledge, there is up to date no formal definition of the "realizes" relationship and also no possibility of verification.

The aim of this paper is to close this gap. We propose a formalization of the "realizes" relationship w.r.t. Java implementations. For this purpose we first define the syntactic and semantic requirements induced by a design model that is given by a UML class diagram and associated OCL constraints. The semantical requirements are presented by Hoare formulas which express pre- and postconditions for the method implementations. We verify these requirements using the axioms and rules of a new Hoare calculus for Java-like sequential programs proposed in [20]. Even if in practice such proofs will not be done in full our approach provides a tool for verifying the critical and important parts of a realization relationship.

Our work has been influenced by several other calculi. A Hoare calculus for Java has been proposed by Poetzsch-Heffter and Müller [18, 19]. We see as a main drawback that *loc.cit.* use an explicit representation of state in their calculus which is thus not suited for OCL. Similarly the calculi of Abadi and Leino [1, 15] do not fit to Java and OCL. Our calculus avoids this problem and is directly tuned to OCL. In this sense we follow rather the ideas of Gries and De Boer who handle arrays and references by explicit substitution [10]. Other relevant work on Hoare-calculi includes a calculus of records [7] and treatment of recursion [22].

The JML approach [14] extends the Java language such that programs (including exceptions) can be annotated by specifications. Proof obligations are generated but proofs can only be performed after a translation into a voluminous semantic description of Java that does not make use of a Hoare-logic but is of denotational flavour instead.

Our calculus extends the usual rules for while programs with blocks by rules for update of instance variables – handled by an explicit substitution operator which also takes care of aliasing –, for creation of objects – using a special constant –, for recursive method specifications – taking care of inheritance –, and method calls.

2 The General Method

During the development of complex software systems various documents on different levels of abstraction are produced ranging from analysis models to concrete implementations (in terms of some programming language code). In this paper we focus on the (formal) relationship between system design and implementation.

2.1 The Design Model

Following the Unified Process (cf. [21]) a design model can be presented by a design subsystem. We assume that the subsystem contains classes, inheritance and association relations such that any association is directed and equipped with a role name and a multiplicity at the association end. As an essential ingredient of our approach the elements of a design model will be equipped with OCL constraints for specifying properties like invariants of classes and pre- and postconditions for the operations.

In the following we consider as an example the design model for a (simple) account subsystem of a bank application shown in Fig. 1. For any checking account there is a history which stores the amounts of all deposit operations performed on the account. To describe precisely the desired effects of the operations in terms of pre- and post-conditions we use OCL-constraints which also include appropriate invariants for the specialized account classes (cf. Table 1).

Fig. 1. Design Model for Accounts

2.2 The Implementation Model

An implementation model is given by an implementation subsystem (in the sense of [21]) which contains components that may be related by dependency relations. In our approach any component C.java will be a Java file containing the code of a Java class C. We assume that all attributes of Java classes are declared "private" or "private protected" to ensure encapsulation of object states. The code of the implementation model is shown in Table 1.

2.3 The Realization Relation

A realization relation connects a given design model and its corresponding implementation model as shown in Fig. 2. We say that the realization relation between "MyDesignSubsystem" and "MyJavaSubsystem" holds if the following syntactic and semantic requirements are satisfied.

Syntactic requirements: First the classes occurring in the design subsystem have to be mapped to components of the implementation model. This can be done by using trace dependencies as considered in [21]. We require that every class C of the design model is related by a trace dependency to a Java component C.java

context Account::deposit(n:Integer)
pre: $n \geq 0$
post: bal = bal@pre + n
	`abstract class Account`
	`{  private protected int bal;`
	`   abstract void deposit(int n) {}`
	`}`

context SavingsAccount
inv: bal ≥ 0 and interestRate ≥ 0
	`class SavingsAccount extends Account`
	`{  private int interestRate;`
	
	`  public void deposit(int n)`
	`  {  this.bal = this.bal + n;`
	`  }`

context SavingsAccount::
addInterest()
post: bal = bal@pre +
bal@pre * interestRate/100
	`  public void addInterest()`
	`  {  int interest = this.bal *`
	`              this.interestRate/100;`
	`     this.deposit(interest);    }`
	`}`

context CheckingAccount
inv: chargeRate ≥ 0
	`class CheckingAccount extends Account`
	`{  private int     chargeRate;`
	`   private History history;`

context CheckingAccount::
deposit(n: Integer)
pre: $n \geq 0$
post: history.oclIsNew and
history.amount = n and
history.history = history@pre
	`   public void deposit(int n)`
	`   {  this.bal = this.bal + n;`
	`      History h = new History();`
	`      h.amount = n;`
	`      h.history = this.history;`
	`      this.history = h;        }`
	`}`

	`class History`
	`{  private int     amount;`
	`   private History history;`
	`}`

Table 1. OCL-Constraints and implementation model for Account Subsystem

as depicted in Fig. 2. The trace dependency between C and C.java is supposed to hold if the following conditions are satisfied:

1. Each attribute of the design class C is also an attribute of the Java class C and for each role name at the end of a directed association the Java class contains a corresponding reference attribute with the same name. (Note that standard types may be slightly renamed according to the Java syntax and that role names with multiplicity greater than one map to reference attributes of some container type.)
2. For each operation m specified in the design class C there is a method declaration in the Java class C and vice versa (up to an obvious syntactic modification of the signature). The operation m of the design model has the property { abstract } iff the method m is an abstract method.
3. The design class C is a (direct) subclass of a design class A iff the Java class C extends the Java class A.

Fig. 2. Trace Dependency and Realization Relation

These conditions guarantee (in particular) that the OCL expressions used as constraints for the design model can be interpreted in the implementation model which is necessary to define the semantical requirements. Moreover, note that the above conditions are satisfied by usual code generators for Java classes from UML class diagrams.

Semantic requirements: Let us first stress that the semantic requirements considered in the following are derived solely from the OCL constraints attached to the design model. This means that constraints imposed by the UML class diagram itself (like multiplicities or { query } properties of operations) and any kind of frame assumption will not be considered here if not explicitly expressed by an OCL constraint.

For the formulation of the semantic requirements we assume that the syntactic requirements from above are satisfied. Let us first discuss the role of invariants. According to [23] an invariant *INV-C* defined in the context of a class C means that *INV-C* evaluates to true for all instances of C at any moment of time. Since, by assumption, all attributes occurring in an implementation model are private or private protected the state of an object can only be modified by method invocations. Therefore the basic idea is to require that the invariant is preserved by any method invocation[1] for objects of C and that the invariant holds also for any object of C after its creation. These conditions, however, are not sufficient if there is a superclass A of C which has also an associated invariant, say *INV-A*. Then, in order to satisfy Liskov's substitution principle for subtypes [16], *INV-A* should be inherited by C. Hence, in general, we have to consider for any class C the conjunction of *INV-C* and all invariants *INV-A* associated to a superclass A of C. For any design class C, this conjunction will be denoted in the following by *INV-conj-C*. [2]

For dealing with object creation we transform any post-condition *POST* occurring in the design model into the expression *POST+* where any occurrence of an OCL expression "t.oclIsNew" with some term t of some type C is replaced by "t.oclIsNew and *INV-conj-C*[$t/this$]".

Having the above definitions in mind we require that for each design class C the following conditions are satisfied:

[1] For simplicity, we assume that all methods are public. Otherwise the approach could be easily extended to take into account UML visibility markers in the design model which then should be preserved by the trace dependency.

[2] Obviously, if *INV-C* is stronger than *INV-A* for any superclass A then *INV-conj-C* is equivalent to *INV-C*.

1. Pre- and post-conditions associated to operations of C are respected by corresponding method implementations. This means that for each operation m specified in the design class C with OCL-constraint

   $$\text{context } C\text{:: m}(\mathrm{p}_1 : \mathrm{T}_1, \ldots, \mathrm{p}_n : \mathrm{T}_n)\text{pre} : \mathit{PRE} \text{ post} : \mathit{POST}$$

   the given Java subsystem satisfies the Hoare formula

   $$\{\mathit{PRE} \text{ and } \mathit{INV\text{-}conj\text{-}C}\}\ C\texttt{::m(}\mathrm{p}_1 : \mathrm{T}_1, \ldots, \mathrm{p}_n : \mathrm{T}_n\texttt{)}\ \{\mathit{POST+}\}$$

   where C denotes the Java class with method m defined in the component C.java. The formal basis of this proof obligation will be provided in the next sections. In particular, according to Definitions 8 and 9, the satisfaction of the above Hoare formula means that any method body of m provided in C or in a subclass C' of C (which eventually overrides m) respects the given pre- and post-condition. Thus Liskov's substitution principle is satisfied.
   Note that it may also be the case that in the design model there is a subclass C' of the design class C which redefines m in the sense that it provides an additional OCL constraint with pre- and post-conditions PRE' and POST' for m. In this case the realization relation requires that both Hoare formulas

   $$\{\mathit{PRE} \text{ and } \mathit{INV\text{-}conj\text{-}C}\}\ C\texttt{::m(}\mathrm{p}_1 : \mathrm{T}_1, \ldots, \mathrm{p}_n : \mathrm{T}_n\texttt{)}\ \{\mathit{POST+}\}$$
   $$\{\mathit{PRE}' \text{ and } \mathit{INV\text{-}conj\text{-}C'}\}\ C'\texttt{::m(}\mathrm{p}_1 : \mathrm{T}_1, \ldots, \mathrm{p}_n : \mathrm{T}_n\texttt{)}\ \{\mathit{POST'+}\}$$

   must be satisfied by the Java subsystem. For instance, the pre- and post-conditions in our example lead to the proof obligations (1-3) of Table 2.
2. Invariants are preserved by method implementations. This means that for each operation m specified in the design class C or in a superclass of C the given Java subsystem satisfies the Hoare formula

   $$\{\mathit{PRE} \text{ and } \mathit{INV\text{-}conj\text{-}C}\}\ C\texttt{::m(}\mathrm{p}_1 : \mathrm{T}_1, \ldots, \mathrm{p}_n : \mathrm{T}_n\texttt{)}\ \{\mathit{INV\text{-}conj\text{-}C}\}$$

   where PRE denotes the pre-condition required for m (if any). For instance, considering the invariants of the account example we obtain the proof obligations (4-6) of Table 2.

3 OCL$^{\text{light}}$

OCL$^{\text{light}}$ is a representative kernel of OCL which should be easily extendible to full OCL. Yet, it deliberately differs from OCL in some minor syntactic points explained below.

3.1 Syntax

OCL$^{\text{light}}$ admits the use of so-called "logical variables" for eliminating expressions of the form "t@pre" from post-conditions. Such variables cannot be altered

$\{\text{n} \geq 0\}$ 1) `Account::deposit(n:Integer)` $\{\text{bal} = \text{bal@pre} + \text{n}\}$	$\{\text{n} \geq 0 \text{ and bal } \geq 0 \text{ and interestRate} \geq 0\}$ 4) `SavingsAccount::deposit(n:Integer)` $\{\text{bal } \geq 0 \text{ and interestRate} \geq 0\}$

$\{\text{n} \geq 0 \text{ and chargeRate} \geq 0\}$
2) `CheckingAccount::`
`deposit(n:Integer)`
$\{$history.oclIsNew and
history.amount = n and
history.history = history@pre$\}$
	$\{\text{bal} \geq 0 \text{ and interestRate } \geq 0\}$
	5) `SavingsAccount::addInterest()`
	$\{\text{bal} \geq 0 \text{ and interestRate} \geq 0\}$

$\{\text{true}\}$ 3) `SavingsAccount::addInterest()` $\{\text{bal} = \text{bal@pre}+$ $\text{bal@pre} * \text{interestRate}/100\}$	$\{\text{n} \geq 0 \text{ and chargeRate} \geq 0\}$ 6) `CheckingAccount::deposit(n:Integer)` $\{\text{chargeRate} \geq 0\}$

Table 2. Proof obligations for the account example

by any program. All other variables are simply referred to as "program variables". By contrast to Table 2, we stipulate that all instance variables are fully qualified, i.e. we write "this.bal" instead of just "bal".

Note that formal parameters of methods are assumed to appear as *logical variables* in assertions since they are not allowed to change (call-by-value). Moreover, we use "this" and "null" although the former is called "self" in OCL and the latter is expressed in OCL by the use of the formula "isEmpty", i.e. instead of "t→isEmpty" write "t = null". The OCL-term-syntax is extended by an operation "new(C)". It should not be used in OCL- specifications, but it may pop up in assertions during the verification process to cope with object creation (cf. Section 5.3). that is sound w.r.t. the above given interpretation function.

General OCL$^{\text{light}}$*-terms* may additionally be built from

$$\begin{array}{rll} t ::= & \langle \mathit{Var}\rangle.\text{a@pre} & \text{field variables in previous state} \\ \mid & \langle \mathit{Var}\rangle.\text{a.oclIsNew} & \text{test for new field variable} \end{array}$$

where $\langle \mathit{Var}\rangle$ must not be a logical variable.

OCL$^{\text{light}}$-*formulas* are expressions of type bool subsuming equality, forall, exists, and includes-expressions.

Notation: Usually we use capital letters (X, Y, Z) for logical variables and small ones for program variables. An exception from the rule are the formal parameters of methods which are uniquely identified by syntax and thus can remain lowercase although regarded logical.

3.2 Semantics of OCL$^{\text{light}}$-terms

There is an interpretation function, $[\![\_]\!]_{\_,\_,\_,\_}$, taking a *pure* OCL$^{\text{light}}$-term, a store (containing the objects), a (runtime-) stack (containing actual parameters of methods and local variables), two environments – one for logical variables and one for query names –, and yields an element in a semantic domain. The definition of $[\![e]\!]_{\mu,\sigma,\beta,\rho}$ is by induction on e. It is rather straightforward and thus omitted (it can be found in [20]). However, query calls were not considered in *loc.cit.*, therefore we have introduced an environment for queries and the semantics of a call for query q is as follows:

$$[\![e_0.\mathrm{q}(\boldsymbol{e})]\!]_{\mu,\sigma,\beta,\rho} = \rho(\mathrm{q})([\![e_0]\!]_{\mu,\sigma,\beta,\rho}, [\![\boldsymbol{e}]\!]_{\mu,\sigma,\beta,\rho}, \mu)$$

where a query environment ρ maps a query name to a function $\rho(q)$, taking as input an object reference (the actual value of *this*), some arguments of a type determined by the argument types of the query and a store (which is needed to obtain the field values of *this*). Moreover, $[\![\boldsymbol{e}]\!]$ is the canonical generalisation of $[\![\_]\!]$ to a list of terms. In the following we will analogously use extensions of β and σ to lists of variables.

The axiomatization of the OCL$^{\text{light}}$-logic contains the usual axioms for natural numbers, typed finite set theory, and booleans. The "forall" and "exists" quantifiers are always bounded by a set. The two "non-standard" operations are "new(C)" representing a free object reference, and "allInstances" referring to all actually existing and valid objects of a certain class type. They are axiomatized as follows:

$$\begin{gathered}\mathrm{not}(\mathrm{new}(C) = \mathrm{null})\\ C.\mathrm{allInstances}{\rightarrow}\mathrm{includes}(t) \text{ iff } \mathrm{not}(t = \mathrm{null}) \text{ and } \mathrm{not}(t = \mathrm{new}(C))\end{gathered}$$

where t is of type C, "iff" means "if, and only if" obtained from "implies". Since not(C.allInstances → includes(t) = C.allInstances → includes(t')) implies not($t = t'$) one can derive e.g. C.allInstances → forall(Y|not(new(C) = Y)).

The queries need a bit of work too. If q is a query with precondition P and postcondition Q the following axiom is supposed to hold:

$$P[e_0/\mathrm{this}, \boldsymbol{e}/\boldsymbol{p}] \text{ implies } Q[e_0/\mathrm{this}, \boldsymbol{e}/\boldsymbol{p}, e_0.q(\boldsymbol{e})/\mathrm{result}]$$

This axiom is only sound, of course, if ρ(q) obeys the specification given as pre- and post-condition which will be generally assumed in the following, i.e. we require for any ρ that for any OCL-query-specification in a set of class declarations D, context q($\boldsymbol{p} : \boldsymbol{\tau}$) : τ_r pre : P post : Q, it holds that

$$\forall \mu, \sigma, \beta.\ [\![P_q]\!]_{\mu,\sigma,\beta,\rho} = \mathit{true} \text{ implies } [\![Q_q]\!]_{\mu,\sigma[\mathit{result} \mapsto \rho(q)(\sigma(\mathit{this}),\beta(\boldsymbol{p}),\mu)],\beta,\rho} = \mathit{true}$$

abbreviated to $\rho \Vdash \mathrm{Queries}(D)$.

Note that formal parameters are treated as logical variables. This is justified by the assumption that we only deal with call-by-value parameter-passing.

Theorem 1. (Soundness) *There is an axiomatization of the OCL$^{\text{light}}$-logic with pure terms, $\vdash_l$, such that for any pure OCL$^{\text{light}}$-formula Q it holds that*

$$\vdash_l Q \;\Rightarrow\; \forall \mu, \sigma, \beta, \rho.\ \rho \Vdash \mathrm{Queries}(D) \text{ implies } [\![Q]\!]_{\mu,\sigma,\beta,\rho} = \mathit{true}$$

For general OCL$^{\text{light}}$-terms the interpretation function has an additional parameter representing the *old* store, i.e. the interpretation function is written $\llbracket e \rrbracket^{\mu_p}_{\mu,\sigma,\beta,\rho}$ where μ_p denotes the old store, whereas μ stands for the actual one.

Definition 2. The interpretation function for *general* OCL$^{\text{light}}$-terms is defined inductively such that $\llbracket t\text{@pre} \rrbracket^{\mu_p}_{\mu,\sigma,\beta,\rho} = \llbracket t \rrbracket_{\mu_p,\sigma,\beta,\rho}$ and $\llbracket t.\text{oclIsNew} \rrbracket^{\mu_p}_{\mu,\sigma,\beta,\rho} = (\llbracket t \rrbracket_{\mu,\sigma,\beta,\rho} \notin \mu_p)$ where $x \notin \mu_p$ is true iff x is not referring to an existing object in μ_p. All other cases follow literally the interpretation for the pure case.

3.3 Transformation of OCL$^{\text{light}}$-formulas

Postconditions may contain expressions of the form "t.a@pre" and "t.oclIsNew" which are forbidden in preconditions. This is impractical in proofs of Hoare-formulas where the postcondition of one statement may appear as precondition of another statement. Therefore we introduce logical variables for encoding the effects of "@pre" and "oclIsNew".

Definition 3. For a pair of general OCL$^{\text{light}}$-formulas (P, Q) we define the syntactic transformation $(P, Q)^* = (P^*, Q^*)$ as follows:

$$P^* = (P\,\text{and}\,t_i = X_i\,\text{and}\,C_j.\text{allInstances} = A_j)$$
$$Q^* = Q[X_i/t_i\text{@pre}][\text{not}(e_j = \text{null})\,\text{and}\,\text{not}(A_j \rightarrow \text{includes}(e_j))/e_j.\text{oclIsNew}]$$

where $\{t_i\text{@pre} \mid i \in I\}$ contain all occurrences of "@pre"-variables in Q and $\{e_j.\text{oclIsNew} \mid j \in J\}$ contain all occurrences of "oclIsNew" in Q. The C_j are the class types of the e_j. All X_i and A_j are new logical variables *not* occurring in P or Q.

Example 4. We can transform the pre- and postcondition of the `deposit` operation in `Account` to the following Hoare-formula:

{ this.bal = M and n ≥ }`Account::deposit(Integer n)`{ this.bal = M + n }

The "oclIsNew" part of the pre-/postcondition of deposit in CheckingAccount is transformed as follows (written vertically):

{ History.allInstances = H and n ≥ 0 }
`CheckingAccount::deposit(Integer n)`
{ not(this.history = null) and not(H→includes(this.history)) and ... }

3.4 Correctness of the Transformation

Proposition 5. *Let P, Q be general OCL$^{\text{light}}$-formulas and $(P, Q)^* = (P^*, Q^*)$. Then for all $\mu, \mu_p, \sigma, \beta$ and ρ we have*

$$\llbracket P \rrbracket_{\mu_p,\sigma,\beta,\rho} \Rightarrow \llbracket Q \rrbracket^{\mu_p}_{\mu,\sigma,\beta,\rho} \text{ iff } \llbracket P^* \rrbracket_{\mu_p,\sigma,\beta^*,\rho} \Rightarrow \llbracket Q^* \rrbracket_{\mu,\sigma,\beta^*,\rho}$$

$$\textit{where} \qquad \beta^*(Z) = \begin{cases} \llbracket t_i \rrbracket_{\mu_p,\sigma,\beta,\rho} & \text{if} \quad Z \equiv X_i, i \in I \\ \llbracket C_j.\text{allInstances} \rrbracket_{\mu_p,\sigma,\beta,\rho} & \text{if} \quad Z \equiv A_j, j \in J \\ \beta(Z) & \text{otherwise} \end{cases}$$

and $(t_i)_{i\in I}$ *and* $(C_j)_{j\in J}$ *are as in Def. 3.*

4 Java$^{\text{light}}$

The object-oriented programming language of discourse is supposed to be a subset of sequential Java with methods and constructors without exceptions.

There are some restrictions, however, on the syntax that deserve explanation. First, we do not allow arbitrary assignments Exp`.a =` Exp as we will only be able to define substitution for instance (field) variables $x.a$ where x is a local variable or a formal parameter (or `this`). This is, however, no real restriction as for an assignment e`.a =`e' one can also write x`=`e`;` x`.a =`e'. This sort of a decomposition of compound expressions is well known from compiler construction. Second, we distinguish a subset of expressions without side-effects (*Exp*) and with possible side-effects (*Sexp*). The first forms a proper subset of OCL$^{\text{light}}$-expressions and can thus be substituted for (instance) variables. This is why all the arguments of a method call must be side-effect free. The restricted syntax for expressions is still sufficient since, again, one can decompose any expression using auxiliary variables.

In general, dealing with partial correctness only, we shall only consider verification of programs that are syntax and type correct. For technical simplicity two minor simplifications of the Java type-system are in use. We ignore shadowing of field variables and method overloading (by different number and types of argument variables).

Semantics For the purpose of this paper it is sufficient to treat the operational semantics of Java$^{\text{light}}$ abstractly.

Definition 6. An operational semantics for Java$^{\text{light}}$ is a family of partial functions

$$(\mathcal{T}^C)_{C\in \mathit{Classname}} : \mathit{JavaL} \times \mathit{Store} \times \mathit{Stack} \rightharpoonup \mathit{Store} \times \mathit{Stack}$$

that is defined – assuming that *this* has actual type C – only if execution of the Java-program terminates successfully. Moreover, the result has to be in accordance with the requirements of the Java Specification [12]. The restriction $\mathcal{T}_k^C$ yields the same result as $\mathcal{T}^C$ if the evaluation depth (the call-stack-depth) of the computation is less than k; otherwise it is undefined.

This restriction is necessary to give a sound interpretation to specifications of recursive methods (see also [22, 20]).

A possible operational semantics that fits can be found in [8, 9, 2]).

5 Hoare Calculus

In this section we present a Hoare calculus for Java$^{\text{light}}$ with assertions written in *pure* OCL$^{\text{light}}$. This calculus extends the well-known Hoare calculi one can find in any textbook (see e.g. [4] or the original text [13]) by a few rules covering assignment to instance variables, object creation (see also [6, 7]), method call, and method specification (inheritance).

5.1 Syntax

Because object-oriented programs are structured by means of classes which in turn break down to fields and methods, we introduce two different Hoare-like judgements, where the one for methods is considered as a special abbreviation:

Definition 7. We first distinguish between two types of Hoare-triples, those *for statements* $\{P\}\ S\ \{Q\}$ and those for *methods* (also called method specifications) $\{P\}\ C :: \mathtt{m}(\boldsymbol{p} : \boldsymbol{\tau})\ \{Q\}$ where S is a Java$^{\text{light}}$-statement, C is a class type, P and Q are pure OCL$^{\text{light}}$ formulas. For method specifications, all program variables appearing in Q must be "this" or "result". Recall that the formal parameters $\boldsymbol{p}$ are assumed to appear as *logical* variables since we assume a call-by-value parameter mechanism. The judgments of the Hoare calculus are then as follows:

1. *Derivable Statement Triples*
   $\Gamma \vdash^C_D \{P\}\ S\ \{Q\}$ where Γ denotes a context being empty or consisting of one method specification, D is the whole set of declarations, i.e. the complete Java-package of discourse, and C is the assumed class type of `this` (which may not be uniquely derivable from the statement S alone).
2. *Derivable Method Triples*
   $\vdash_D \{P\}\ C :: \mathtt{m}(\boldsymbol{p} : \boldsymbol{\tau})\ \{Q\}$ where C and D are as above.

The context for Hoare triples is necessary for the treatment of recursive method specifications. For *mutual* recursive methods the context must be generalized to sets of method triples.

We omit the indices D and C if they can be derived from the context.

5.2 Semantics

The following definition of validity of triples holds for *general* OCL$^{\text{light}}$-assertions P and Q.

Definition 8. Let $\mathcal{T}$ denote a semantic function for Java$^{\text{light}}$. Then Hoare-triples are said to hold relatively to evaluation depth smaller than k if the following holds:

1. *Statement Triples* (partial correctness of statements)
   $\models^{D,C}_k \{P\}\ S\ \{Q\}$, if for any μ, σ, β we have
   $$[\![P]\!]_{\mu,\sigma,\beta,\rho} = \mathit{true} \wedge \mathcal{T}^C_k(S, \mu, \sigma) = (\mu', \sigma') \Rightarrow [\![Q]\!]^{\mu}_{\sigma',\mu',\beta,\rho} = \mathit{true}$$
   where ρ is defined as follows for any query of D defined in class C_q:
   $$\rho(q)(o, \boldsymbol{a}, \mu) = (\mathcal{T}^{C_q}_k(\mathrm{body}(C_q, q, D), \mu, \emptyset[\mathit{this} \mapsto o][\boldsymbol{p} \mapsto \boldsymbol{a}])_1(\mathit{result})$$
2. *Method Triples* (partial correctness of methods)
   $\models^D_k \{P\}\ C\mathtt{::m}(\boldsymbol{p} : \boldsymbol{\tau}\mathtt{)}\ \{Q\}$ if $\forall C' \leq C.\ \models^{D,C'}_k \{P\}\ \mathrm{body}(C', \mathtt{m}, D)\ \{Q\}$ where $\mathrm{body}(C', \mathtt{m}, D)$ is the body of `m` defined in class C' of program package D. If C' just inherits `m` from some superclass C'' then $\mathrm{body}(C', \mathtt{m}, D) = \mathrm{body}(C'', \mathtt{m}, D)$.

Note that it is not clear *a priori* that $\rho \Vdash \text{Queries(D)}$, but it will follow from the proof of $\vdash_D \{P\}\ C\texttt{::q(...)}\ \{Q\}$ for each query q with pre-condition P and post-condition Q.

Definition 9. A triple T is valid in a context Γ, i.e. $\Gamma \models^{D,C} T$, iff

$$\forall k \in \mathbb{N}.\ \models^D_k \Gamma \Rightarrow \models^{D,C}_{k+1} T$$

5.3 Inductive definition of the Hoare calculus

In this section we present the rules (i.e. the calculus) for deriving *correct* specifications for Java$^{\text{light}}$ programs in a purely syntactic way. The rules and axioms below define inductively a relation $\vdash^D_C$, i.e. the derivable statement specifications.

To this end we may make use of the axioms and rules for the OCL$^{\text{light}}$-language (i.e. $\vdash_l$, cf. Theorem 1) and of the "classical" rules of the Hoare calculus for While-languages which are not repeated here (cf. [13, 3]).

In the following we present the rules that deal with object-oriented features.

Field Assignment

$$\{P[e/x.a]\}\ x\texttt{.a=}e\ \{P\} \qquad e \in \mathit{Exp} \qquad \text{(Field variable assignment)}$$

where $t[e/x.a]$ is the substitution for field variables defined inductively as follows:

Definition 10. Define $e'[e/x.a]$ by structural induction on e', the only interesting non-trivial case being (in other cases just push substitution through):

$$(t.b)[e/x.a] \triangleq \begin{cases} t[e/x.a].b & \text{if } b \neq a \\ \text{if } (t[e/x.a] = x) \text{ then } e \text{ else } t[e/x.a].b & \text{otherwise} \end{cases}$$

Example 11. The following Hoare-triple is an instance of the field variable assignment axiom for the body of the `deposit` operation in `SavingsAccount`:

{ (if (this = this) then (this.bal + n) else this.bal) = M + n}
`this.bal = this.bal + n`
{ this.bal = M + n }

which by the Weakening Rules reduces to

$$\{\text{this.bal} + \text{n} = \text{M} + \text{n}\}\ \texttt{this.bal = this.bal + n}\ \{\text{this.bal} = \text{M} + \text{n}\}$$

Again by weakening we obtain the correctness of the body of the method `deposit` of class `SavingsAccount` w.r.t. the transformed OCL$^{\text{light}}$-pre/post-condition of `deposit` given in the superclass `Account` (cf. (1) of Table 2).

Object creation Let $Q[\delta_{C.a}/x.\boldsymbol{a}]$ abbreviate the simultaneous substitution of all field variables $x.a_i$ occurring in Q by a default value of appropriate type. This default value has to be the one that Java$^{\text{light}}$ uses for initialisation (e.g. 0 for integers and null for class types).

$$\begin{array}{ll}\{Q[\delta_{C.a}/x.\boldsymbol{a}][\text{new}(C)/x, C.\text{allInstances}{\rightarrow}\text{including}(\text{new}(C))/C.\text{allInstances}]\} & \\ \texttt{x=new}\,C\texttt{()} & \text{(new)} \\ \{Q\} & \end{array}$$

where Q does not contain any query calls nor new(C).

Recall that "new(C)" and query calls can be eliminated using the consequence rule of standard Hoare calculus and the axioms mentioned in Section 3.2.

Example 12. The correctness of `deposit` in `CheckingAccount` involves proving the following Hoare-formula (*):

{ H = History.allInstances }
`History h = new History()`
{ not($H{\rightarrow}$includes(h)) and not(h = null) }

Using the axiom for object creation the derived precondition is

$$(**) \qquad \text{not}(H{\rightarrow}\text{includes}(\text{new(History)})) \text{ and not}(\text{new(History)} = \text{null})$$

Because of the axioms for "new(History)" and "History.allInstances" the precondition of (*) implies (**). Thus by the weakening rule, the Hoare-formula (*) is proven.

return-statement Returning a value means assigning it to variable *result.*

$$\{Q[e/\text{result}]\}\ \texttt{return}\ e\ \{Q\} \qquad \text{(return)}$$

Method specifications The partial correctness of a method specification for `m` in class C can be derived from the partial correctness of all bodies of `m` in C and in any subclass of C where for dealing with recursion the partial correctness of the method specification can be assumed.

$$\frac{\forall C' \leq C.\, \{P\}\ C\texttt{::m(}\boldsymbol{p}\texttt{:}\boldsymbol{\tau}\texttt{)}\ \{Q\} \vdash^{D}_{C'} \{P\}\ \text{body}(C', \texttt{m}, D)\ \{Q\}}{\{P\}\ C :: \texttt{m}(\boldsymbol{p} : \boldsymbol{\tau})\ \{Q\}} \quad \text{(MethodSpec)}$$

Example 13. By proving the correctness of the method bodies of `deposit` in `SavingsAccount` (cf. Example 11) and `CheckingAccount` i.e.

$\vdash^{\text{AccountJavaSubsystem}}_{\text{SavingsAccount}}$ { this.bal = M and n $\geq$ 0 }
body(SavingsAccount,`deposit`,AccountJavaSubsystem)
{ this.bal = M + n } and
$\vdash^{\text{AccountJavaSubsystem}}_{\text{CheckingAccount}}$ { this.bal = M and n $\geq$ 0 }
body(CheckingAccount,`deposit`,AccountJavaSubsystem)
{ this.bal = M + n }

we conclude the correctness of `deposit` w.r.t. its transformed OCL-constraint by rule (MethodSpec).

$$\vdash_{\text{AccountJavaSubsystem}} \begin{array}{l}\{\ \text{this.bal} = \text{M and n} \geq 0\ \} \\ \texttt{Account::deposit(n : Integer)} \\ \{\ \text{this.bal} = \text{M} + \text{n}\ \}\end{array}$$

Method Call The rules for the method call must take into consideration the method dispatch of the programming language. This is ensured by using the method specification in the premise.

$$\frac{\{P\}\ \mathit{type}(e_0)\texttt{::m(}\boldsymbol{p}\texttt{:}\boldsymbol{\tau}\texttt{)}\ \{Q\} \qquad \vdash_l Q[e_0/\text{this}]\,\text{implies}\,R[\text{result}/x]}{\{P[e_0/\text{this}]\ \text{and}\ \boldsymbol{p} = \boldsymbol{e}\}\ x\texttt{=}e_0\texttt{.m(}\boldsymbol{e}\texttt{)}\ \{R\}} \quad \text{(Call)}$$

Note that one cannot simplify the rule by dropping the implication in the hypothesis and changing the postcondition in the conclusion to $Q[e_0/\ \mathit{this}, x/\mathit{result}]$ since this would blur the distinction between x before and after execution of the method call and thus lead to an unsound rule. For the very same reason the arguments $\boldsymbol{e}$ cannot be substituted into Q.

Logical variables can be replaced by special side-effect free expressions.

$$\frac{\{P\}\ x\texttt{=}e_0\texttt{.m(}\boldsymbol{e}\texttt{)}\ \{Q\}}{\{P[\boldsymbol{e'}/\boldsymbol{Z}]\}\ x\texttt{=}e_0\texttt{.m(}\boldsymbol{e}\texttt{)}\ \{Q[\boldsymbol{e'}/\boldsymbol{Z}]\}} \quad \text{if } \boldsymbol{e'} \in \mathit{Exp},\ x \notin LV(\boldsymbol{e'}), IV(\boldsymbol{e'}) = \emptyset$$

(Call Invariance)

where $LV(\boldsymbol{e'})$ and $IV(\boldsymbol{e'})$ denote the local variables and the instance variables occurring in vector $\boldsymbol{e'}$, respectively, and $\boldsymbol{Z}$ is a vector of logical variables (thus not occurring in any program). The variable conditions ensure that $\boldsymbol{e'}$ is not changed by the method invocation.

For method calls with return type `void` there is an analogous rule.

$$\frac{\{P\}\ \mathit{type}(e_0)\texttt{::m(}\boldsymbol{p}\texttt{:}\boldsymbol{\tau}\texttt{)}\ \{Q\}}{\{P[e_0/\text{this}]\,\text{and}\,\boldsymbol{p} = \boldsymbol{e}\}\ e_0\texttt{.m(}\boldsymbol{e}\texttt{)}\ \{Q[e_0/\text{this}]\}} \quad \text{(CallVoid)}$$

We omit the analogous invariance rule for methods with return type.

Example 14. In the following we prove a property of `deposit` which is used in the proof of the constraint for `addInterest`:

$$\cfrac{\cfrac{\{\text{this.bal} = \text{M and n} \geq 0\}\ \texttt{SavingsAccount::deposit(n:Integer)}\ \{\text{this.bal} = \text{M+n}\}}{\{\text{this.bal} = \text{M and n} \geq 0 \text{ and n=interest}\}\ \texttt{this.deposit(interest)}\ \{\text{this.bal=M+n}\}}\ \text{(MethodCall)}}{\{Q\}\ \texttt{this.deposit(interest)}\ \{\text{this.bal =M+M*I/100}\}}\ \text{(CallInvariance)}$$

where $Q \equiv$ “this.bal = M and M*I/100 $\geq$ 0 and M*I/100=interest” and “I” is a logical variable denoting the value of “this.interestRate”. Note that for proving “M*I/100 $\geq$ 0” we need the invariant of `SavingsAccount` asserting “this.bal $\geq$ 0 and this.interestRate $\geq$ 0”.

5.4 Correctness

Theorem 15. *The presented Hoare calculus for pure OCL$^{\text{light}}$-formulas and Java$^{\text{light}}$ programs is sound w.r.t. the operational semantics of Java$^{\text{light}}$given in [9], i.e.*

$$\Gamma \vdash_D^C \{P\}\ S\ \{Q\} \Rightarrow \Gamma \models^{D,C} \{P\}\ S\ \{Q\}$$

Proof. [20]

Corollary 16. *For general OCL$^{\text{light}}$-formulas P and Q we therefore get*

$$\Gamma \vdash_D^C \{P^*\}\ S\ \{Q^*\} \Rightarrow\ \Gamma \models^{D,C} \{P\}\ S\ \{Q\}$$

Proof. The proof is a consequence of the theorem above and Proposition 5.

Currently we are investigating the completeness of the Hoare calculus. It appears that we need some additional (admissible) rules such as conjunction introduction and the introduction of existential quantifiers, see e.g. [4].

6 Verifying the Realization Relation

In this section we sketch the proof of the correctness of the realization relation of the AccountSubsystem (see Fig. 3). According to the definition in Section 2.3 we have to show the trace dependencies, the satisfaction of the pre-/postcondition constraints and the preservation of the OCL-invariants.

Fig. 3. Realization relation of the AccountSubsystem

Trace dependencies The trace dependencies are obviously satisfied: for each class of AccountSubsystem there exists a corresponding Java class in AccountJavaSubsystem so that the attributes, methods and inheritance relations are preserved.

Satisfaction of pre-/postconditions The proof obligations (1-3) of Table 2 shown in Section 2.3 have to be verified. For this purpose, according to Corollary 16, it is sufficient to consider their transformations which can be proved as sketched in Example 13 (for (1)), Example 12 (for (2)), and Example 14 (for (3)).

Preservation of invariants The AccountSubsystem contains invariants for SavingsAccount and CheckingAccount. It is easy to prove the associated conditions (4-6) shown in Section 2.3.

7 Concluding Remarks

We have presented a new formal approach for verifying the realization of UML design models by Java subsystems and a new Hoare calculus for a sequential subset of Java and a subset of OCL as assertion language. This is a first step towards the goal of providing a basis for formal software development with UML. But one can see several necessary extensions of our approach, for the UML part as well as for the Hoare calculus. In this paper we have restricted the design models to classes and their relationships. In the following we plan to consider also interfaces. Here, our approach of [5] where we propose a constraint language for interfaces may provide a good basis for the extension. Another important question concerns the composability of subsystems: Under which conditions is the correctness of the realizes relationship preserved if two subsystems with correct realizations are composed? Concerning the Hoare calculus it should be easy to extend semantics, calculus, and soundness proof to the full OCL-language (with bags, sequences and many operations on them). In order to analyse the practicability of our calculus we also need to carry out further case studies. Those examples might then propose additional admissible or derived proof-rules for the Hoare calculus in order to support the verification process, i.e. to simplify the reasoning.

Acknowledgement

Thanks to Hubert Baumeister for useful discussions on the realization relation and Alexander Knapp for helping with the conversion of the diagrams.

References

1. M. Abadi and K.R.M. Leino. A logic of object-oriented programs. In Michel Bidoit and Max Dauchet, editors, *Theory and Practice of Software Development: Proceedings / TAPSOFT '97, 7th International Joint Conference CAAP/FASE*, volume 1214 of *Lecture Notes in Computer Science*, pages 682–696. Springer-Verlag, 1997.
2. J. Alves-Foss, editor. *Formal Syntax and Semantics of Java*, volume 1523 of *Lect. Notes Comp. Sci.* Springer, Berlin, 1999.
3. K.R. Apt. Ten Years of Hoare's Logic: A Survey – Part I. *TOPLAS*, 3(4):431–483, 1981.
4. K.R. Apt and E.R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer, 1991.
5. M. Bidoit, R. Hennicker, F. Tort, and M. Wirsing. Correct realizations of interface constraints with OCL. In *UML'99, The Unified Modeling Language - Beyond the Standard, Fort Collins, USA*, volume 1723 of *Lecture Notes in Computer Science.* Springer-Verlag, 1999.
6. R. Bornat. Pointer aliasing in Hoare logic. In *Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 102–126. Springer-Verlag, 2000.
7. C. Calcagno, S. Ishtiaq, and P.W. O'Hearn. Semantic analysis of pointer aliasing, allocation and disposal in Hoare logic. In *Principles and Practice of Declarative Programming.* ACM Press, 2000.

8. P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. From Sequential to Multi-Threaded Java: An Event-Based Operational Semantics. In M. Johnson, editor, *Proc. 6$^{\text{th}}$ Int. Conf. Algebraic Methodology and Software Technology*, volume 1349 of *Lect. Notes Comp. Sci.*, pages 75–90, Berlin, 1997. Springer.
9. P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An Event-Based Structural Operational Semantics of Multi-Threaded Java. In Alves-Foss [2], pages 157–200.
10. F.S. de Boer. A WP-calculus for OO. In W. Thomas, editor, *Foundations of Software Science and Computations Structures*, volume 1578 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.
11. D. D'Souza and A.C. Wills. *Objects, components and frameworks with UML: the Catalysis approach*. Addison–Wesley, Reading, Mass., etc., 1998.
12. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison–Wesley, Reading, Mass., 1996.
13. C.A.R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12:576–583, 1969.
14. Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In *Object Oriented Programming: Systems, Languages and Applications (OOPSLA)*, volume 35, pages 208–228. ACM SIGPLAN Notices, 2000.
15. K. Rustan M. Leino. Ecstatic: An object-oriented programming language with an axiomatic semantics. Technical Report KRML 65-0, SRC, 1996.
16. B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.
17. Object Management Group. Unified Modeling Language – Object Constraint Language Specification. Technical report, available at http://www-4.ibm.com/software/ad/standards/ad970808_UML11_OCL.pdf, 1998.
18. A. Poetzsch-Heffter and P. Müller. A logic for the verification of object-oriented programs. In R. Berghammer and F. Simon, editors, *Programming Languages and Fundamentals of Programming*, Lecture Notes in Computer Science. Springer-Verlag, 1997.
19. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science. Springer-Verlag, 1999.
20. B. Reus and M. Wirsing. A Hoare-Logic for Object-oriented Programs. Technical report, LMU München, 2000.
21. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison–Wesely, Reading, Mass., etc., 1998.
22. D. von Oheimb. Hoare logic for mutual recursion and local variables. In V. Raman C. Pandu Rangan and R. Ramanujam, editors, *Found. of Software Techn. and Theoret. Comp. Sci.*, volume 1738 of *LNCS*, pages 168–180. Springer, 1999.
23. J. Warmer and A. Kleppe. *The Object Constraint Language*. Addison–Wesley, Reading, Mass., etc., 1999.