A Specification-Based Test Case Generation Method for UML/OCL

Achim D. Brucker[1], Matthias P. Krieger[2,3], Delphine Longuet[2,3], and Burkhart Wolff[2,3]

[1] SAP Research, Vincenz-Priessnitz-Str. 1, 76131 Karlsruhe, Germany
achim.brucker@sap.com
[2] Univ. Paris-Sud, LRI UMR8623, Orsay F-91405*
[3] CNRS, Orsay F-91405
{krieger,longuet,wolff}@lri.fr

Abstract. Automated test data generation is an important method for the verification and validation of UML/OCL specifications. In this paper, we present an extension of DNF-based test case generation methods to class models and recursive query operations on them. A key feature of our approach is an implicit representation of object graphs avoiding a representation based on object-id's; thus, our approach avoids the generation of isomorphic object graphs by using a concise and still human-readable symbolic representation.

Keywords: OCL, UML, test case generation, specification-based testing.

1 Introduction

Automated test data generation is an important application domain for OCL specifications. Instead of verifying concrete code via a Hoare-Calculus for a specific programming language against OCL method contracts—a technique developed in detail for OCL in [9, 11]—test generation can be a more light-weighted (but logically less safe) formal method to reveal errors both in specification and implementations. A particular advantage of black-box testing is that implementations may consist of arbitrary mixtures of (dirty) programming languages.

In this paper, we will adapt existing specification-based testing techniques to UML/OCL, i. e., an object-oriented specification formalism centered around the concept of an object-graph as state, state-transitions described by class-models and state-charts (which we will ignore here), and a type system based on subtyping and inheritance. The work presented here is based on the previous work on a formal UML/OCL semantics [7, 11] and attempts to develop, in contrast to prior works such as [4], a *comprehensive* test-generation method for the complete language and for realistic test-scenarios. Overall, our contribution consists in:

1. the extension of specification-based test generation methods to the world of object-oriented specifications,

* This work was partially supported by the Digiteo Foundation.

2. a *deductive*, theorem-prover based test data generation from OCL specifications including language features such as recursive query-operations, and
3. a particular representation of object-graph classes by our novel concept of an *alias closure*; rather than representing the explicit object graphs we represent the object identity by an equivalence relation.

This paper is written with hindsight to the HOL-TestGen system [10], into which we will implement the technique presented here in a future step. This implies that our technique must fit to the underlying logical framework Isabelle/HOL (an embedding of UML/OCL has been presented in [11]), and that it can be organized into the generation phases of HOL-TestGen.

2 A Gentle Introduction to a Formal OCL 2.2 Semantics

In this section, we briefly present a formal semantics for OCL 2.2 [20], see [7] for details. With respect to the syntax, we use the mathematical notation of HOL-OCL [8] which allows for a concise presentation of OCL constraints.

2.1 Higher-Order Logic

Higher-order Logic (HOL) [12] is a classical logic with equality enriched by total parametrically polymorphic higher-order functions. It is more expressive than first-order logic, e. g., induction schemes can be expressed inside the logic. Pragmatically, HOL can be viewed as a typed functional programming language like Haskell extended by logical quantifiers.

HOL is based on the typed λ-calculus, i. e., the *terms* of HOL are λ-expressions. Types of terms may be built from *type variables* (like α, β, …, optionally annotated by Haskell-like *type classes* as in $\alpha :: \mathit{order}$ or $\alpha :: \mathrm{bot}$) or *type constructors* (like bool or nat). Type constructors may have arguments (as in $\alpha\,\mathrm{list}$ or $\alpha\,\mathrm{set}$). The type constructor for the function space $\Rightarrow$ is written infix: $\alpha \Rightarrow \beta$; multiple applications like $\alpha_1 \Rightarrow (\ldots \Rightarrow (\alpha_n \Rightarrow \alpha_{n+1})\ldots)$ have the alternative syntax $[\alpha_1, \ldots, \alpha_n] \Rightarrow \alpha_{n+1}$. HOL is centered around the extensional logical equality $\_ = \_$ with type $[\alpha, \alpha] \Rightarrow \mathrm{bool}$, where bool is the fundamental logical type. We use infix notation: instead of $(\_ = \_)\ E_1\ E_2$ we write $E_1 = E_2$. The logical connectives $\_ \land \_$, $\_ \lor \_$, $\_ \Rightarrow \_$ of HOL have type $[\mathrm{bool}, \mathrm{bool}] \Rightarrow \mathrm{bool}$, $\neg\_$ has type $\mathrm{bool} \Rightarrow \mathrm{bool}$. The quantifiers $\forall\,\_.\_$ and $\exists\,\_.\_$ have type $[\alpha \Rightarrow \mathrm{bool}] \Rightarrow \mathrm{bool}$. The quantifiers may range over types of higher order, i. e., functions or sets.

The type discipline rules out paradoxes such as Russel's paradox in untyped set theory. Sets of type α set can be defined isomorphic to functions of type $\alpha \Rightarrow \mathrm{bool}$; the definition of the elementhood $\_ \in \_$, the set comprehension $\{\_.\_\}$, $\_ \cup \_$ and $\_ \cap \_$ is then standard.

2.2 Valid Transitions and Evaluations

We recall that OCL expressions form a typed assertion language whose syntactic elements are composed of (a) operators on built-in data structures such as Boolean or collection types like `Set` or `Bag`, (b) operators of the user-defined data-model such as attribute accessors, type-casts and tests, and (c) user-defined, potentially recursive, side-effect-free method calls.

The topmost goal of the formal semantics for OCL expressions is to define the notion of a *valid transition* over states of a system; even concepts like *object invariants* can be derived from this notion. Let σ be a pre-state and σ' a post-state and let ϕ be a Boolean OCL expression, then we write

$$(\sigma, \sigma') \vDash \phi$$

for "the transition from σ to σ' is valid in ϕ." A formula ϕ is valid if and only if its evaluation in the transition $\tau = (\sigma, \sigma')$ yields true. As all types in HOL-OCL are extended by the special element $\perp$ denoting undefinedness, we define formally:

$$\tau \vDash \phi \equiv \left(I[\![\phi]\!]\,\tau = \llcorner\mathrm{true}\lrcorner\right) .$$

In OCL, the evaluation of all expressions can result in an undefinedness element called `invalid` which we will write $\perp$ for short. The test for definedness (`not _ .oclIsInvalid()`) will be written $\partial\ \_$ and is defined by $\partial X \equiv \mathtt{not}\ (X \triangleq \perp)$. Here, $\_ \triangleq \_$ denotes the strong equality, which is a reflexive, symmetric and transitive congruence relation; therefore, the strong equality allows for substituting equals with equals in any OCL expression, even if the expressions are undefined. In contrast, the standard equality in OCL, i. e. $\_ \doteq \_$, is *strict*, which means $x \doteq \perp$ is strongly equal to $\perp \doteq x$ which is strongly equal to $\perp$.

Since all operators of the assertion language depend on the context τ and results can be $\perp$, all expressions can be viewed as *evaluations* from τ to a type $\alpha_\perp$. All types of expressions are of a form captured by the type abbreviation:

$$V(\alpha) = \sigma \times \sigma \Rightarrow \alpha_\perp ,$$

where $\sigma \times \sigma$ stands for the type of a pair of system states (i. e., the type of τ).

2.3 Semantics of Object Invariants and Operation Contracts

The OCL semantics [20, Annex A] uses different interpretation functions for invariants and pre-conditions; instead, we achieve their semantic effect by a syntactic transformation $\__{\mathrm{pre}}$ which replaces all accessor functions $\_ .\,\mathrm{i}$ by their counterparts $\_ .\,\mathrm{i}\,$`@pre`. For example, $(\mathit{self}.\,\mathrm{i} > 5)_{\mathrm{pre}}$ is just $(\mathit{self}.\,\mathrm{i}\,\mathtt{@pre} > 5)$. The operation `_ .allInstances()` is also substituted by its `@pre` counterpart. Thus, we can re-formulate the semantics of the two OCL top-level constructs, invariant specification and method specification, as follows:

$$\begin{aligned} &I[\![\mathtt{context}\ c : C\ \mathtt{inv}\ n : \phi(c)]\!]\tau \equiv \\ &\qquad \tau \vDash (C\ \mathtt{.allInstances()\text{->}forall}(x|\phi(x))) \wedge \\ &\qquad \tau \vDash (C\ \mathtt{.allInstances()\text{->}forall}(x|\phi(x)))_{\mathrm{pre}} \end{aligned} \tag{1}$$

The standard forbids expressions containing `@pre` constructs in invariants or preconditions syntactically; thus, mixed forms cannot arise. Since operations have strict semantics in OCL, we have to distinguish for a specification of an *op* with the arguments $a_1, \ldots, a_n$ the two cases where all arguments are defined (and *self* is non-null), or not. In the former case, a method call can be replaced by a *result* that satisfies the contract, in the latter case the argument is $\bot$:

$$\begin{aligned}
&I[\![\texttt{context}\ C\ :: op(a_1,\ldots,a_n) : T\\
&\qquad \texttt{pre}\ \phi(self, a_1,\ldots,a_n)\\
&\qquad \texttt{post}\ \psi(self, a_1,\ldots,a_n, result)]\!]\tau \equiv \forall s, x_1,\ldots,x_n.\\
&\qquad\qquad\qquad \Delta(s,x_1,\ldots,x_n) \land \tau \vDash \phi(s,x_1,\ldots,x_n)_{\text{pre}}\\
&\qquad\qquad\qquad\quad \longrightarrow \tau \vDash \psi(s,x_1,\ldots,x_n, s.op(x_1,\ldots,x_n))\\
&\qquad\qquad\qquad \land \neg\Delta(s,x_1,\ldots,x_n) \longrightarrow \tau \vDash s.op(x_1,\ldots,x_n) \triangleq \bot
\end{aligned} \tag{2}$$

where $\Delta(s,x_1,\ldots,x_n)$ is an abbreviation for $\tau \vDash s \not\triangleq \texttt{null} \land \tau \vDash \partial\, s \land \tau \vDash \partial\, x_1 \land \ldots \tau \vDash \partial\, x_n$. This definition captures the two cases: if the arguments of an operation are defined and, moreover, *self* is not `null`, the result of a method call must satisfy the specification; otherwise the operation will be strict and return invalid $\bot$. By these definitions an OCL specification, i. e., a sequence of invariant declarations and operation contracts, can be transformed into a set of (logically conjoined) statements which is called the *context* Γ_τ. The *theory* of an OCL specification is the set of all valid transitions $\tau \vDash \phi$ that can be derived from Γ_τ. For the logical connectives of OCL, a conventional Gentzen-style calculus for pairs of the form $\Gamma_\tau \vdash \phi$ can be developed that allows for inferring valid transitions from Γ_τ by deduction (cf. [11]). Due to the inclusion of arithmetic, any calculus for OCL is necessarily incomplete. It is straight-forward to extend our notion of context to multi-transition contexts such as:

$$\Gamma \equiv \{(\sigma,\sigma') \vDash \phi, (\sigma',\sigma'') \vDash \psi\}$$

such that we can reason over systems executing several transitions.

2.4 Strict Operations and Their Role in Reasoning

The OCL standard [20] defines most operations as strict, not just the special case of the strict equality $x \doteq y$ mentioned earlier. Overall, we have the rule

$$f(x_1,\ldots,\bot,\ldots,x_n) \triangleq \bot. \tag{3}$$

A notable exception from this rule are the logical connectives, which are a three-valued strong Kleene-logic; e. g., $\bot$ `andfalse` $\triangleq$ `false` and analogously $\bot$ `or true` $\triangleq$ `true`. Overall, using a three-valued logic is a burden if a simple compilation of OCL to standard automated theorem provers is envisaged. Looking at the wealth of tools (that are specialized for two-valued logics) like Kodkod [22] or Z3 [17], this is perceived as a major drawback of OCL by many.

The methodology of OCL (in particular the strictness of most operations and the fact that most OCL expressions, e. g., invariants, are, by definition, defined) enforces that a reduction to a two-valued representation is always possible; it suffices to apply the case-distinction:

$$\tau \vDash \phi(\bot) \vee (\tau \vDash \partial\ E \wedge \tau \vDash \phi(E)) \tag{4}$$

exhaustively to all sub-expressions E (the $\tau \vDash \phi(\bot)$-parts will either reduce quickly due to Fact 3 to $\tau \vDash \bot$ which is just false or again be subject to Fact 4). The result are formulae of the form: $\tau \vDash \partial\ E_1 \wedge \cdots \wedge \tau \vDash \partial\ E_n \wedge \tau \vDash \phi$ or just $\Delta_\phi \wedge \tau \vDash \phi$ for short. In this form—called Δ-long-form—all implicit definednesses in a valid OCL-formula are made explicit. We call the process of constructing a Δ-long-form Δ-saturation. We do not distinguish between $\{\tau \vDash E_1 \wedge \tau \vDash E_2\} \cup \Gamma$ and $\{\tau \vDash E_1, \tau \vDash E_2\} \cup \Gamma$.

This process can be optimized: if we have, for example, as consequence of an invariant $\tau \vDash f(a) \doteq b$ in our context Γ (meaning that it holds and, thus, evaluates to true), we can infer that $\tau \vDash \partial\, f(a)$ and $\tau \vDash \partial\, b$. From there, we can further infer $\tau \vDash \partial\, a$ (if f is strict). The same holds for the common connective $\tau \vDash X$ `and` Y (but not for _ `or` _). Once that the implicit knowledge on definedness is established, rules of the following form can be applied:

$$\begin{aligned}
\tau \vDash \texttt{not}\ X &= \neg \tau \vDash X && \text{if } \tau \vDash \partial\, X\\
\tau \vDash X\ \texttt{or}\ \ Y &= \tau \vDash X \vee \tau \vDash Y && \text{if } \tau \vDash \partial\, X \text{ and } \tau \vDash \partial\, Y\\
\tau \vDash X\ \texttt{and}\ Y &= \tau \vDash X \wedge \tau \vDash Y && \text{if } \tau \vDash \partial\, X \text{ and } \tau \vDash \partial\, Y\\
\tau \vDash X \doteq Y &= \tau \vDash X \triangleq Y && \text{if } \tau \vDash \partial\, X \text{ and } \tau \vDash \partial\, Y
\end{aligned}$$

By applying this form of equations to Γ and ϕ, we transform them into sets of judgments of the form $\tau \vDash \phi$, i. e., perfect two-valued statements that can be treated by conventional SMT solvers like Z3 (provided that we add an appropriate background theory that axiomatizes the basic operations of the OCL language).

3 Running Example: Linked Lists

In this section, we present a small UML/OCL specification that will serve as a running example for our test case generation technique. We will also discuss the translation of OCL into HOL and discuss the implicit invariants of this example.

3.1 Singly-Linked Lists

Fig. 1 illustrates our running example of a singly-linked list: the list stores integers as data and links between nodes are modeled by an association. As a node does not necessarily need to have a successor, the association end `next` has multiplicity `0..1`. An invariant of the class states that the integers are stored in a descending order in the list. We specify an operation `insert` that adds an integer to the list. The postcondition of the `insert` operation states that the set of integers stored in the list in the post-state is the set of stored integers in the pre-state extended by the argument. For defining the set of integers stored in the list, we separately specify the recursive query operation `contents()`.

```
context Node
inv: next <> null implies i > next.i

context Node::contents():Set(Integer)
  post: result = if next = null
                 then Set{i}
                 else next.contents()->including(i)

context Node::insert(x:Integer)
  post: contents() = contents@pre()->including(x)
```

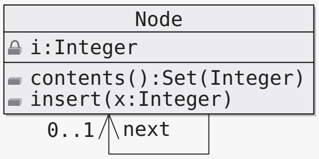

Fig. 1. A Singly-linked list specified in OCL (excerpt)

In the following, we will describe how to build Γ_τ from this OCL specification via the semantic definitions. We will add to Γ_τ semantic presentations of the specification which are already in a "massaged format" suitable for test case generation later. Since the transition is not changing in the rest of this paper, we will assume one global transition τ (understood to be relative to the specification of this example); we will drop the index and abbreviate $\tau \vDash \phi$ to just $\vDash \phi$. In our test case generation approach, we assume that all diagrammatic constraints over the class model are represented as OCL expressions (for details, see [15]). For example, associations are usually represented by collection-valued class attributes together with OCL constraints expressing the multiplicity.

3.2 Translating Invariants into Recursive HOL-Predicates

The example in Fig. 1 only includes one explicit invariant. The multiplicity constraints in the class model constitute invariants semantically. For our example, the multiplicity constraints could be expressed as follows in OCL:

```
inv: (next = null or next <> null) and i <> null
```

In the following, we will assume that attributes and arguments that have a basic datatype (e. g., `Integer`) have a multiplicity of `1..1`, i. e., they cannot be `null`. Thus we can simplify the invariant representing the multiplicity constraints to:

```
inv: (next = null or next <> null)
```

This simplification improves the readability of the formulae in this paper and is not a fundamental restriction of our approach.

For our purposes it will be convenient to convert invariants to recursive predicates and add them to Γ, paving the way for the exploration of input parameters by simply unfolding them rather than making them, based on Fact 1, lengthy arguments over `.allInstances()`. Of course, not any recursive predicate is consistent; however *these* recursive predicates can be derived from the invariants by using a greatest fixed-point construction and proving that the body of the invariant is monotone—the reader interested in the details is referred to HOL-OCL [9] where this is done automatically (albeit for OCL 2.0, i. e., without `null`):

$$\begin{aligned}
&\forall\ \mathtt{self}.\ \ \models \partial\ \mathtt{self} \land \models \mathtt{self} \not\doteq \mathtt{null} \longrightarrow\ \models \mathrm{inv}_{\mathtt{Node}}(\mathtt{self})\\
&\quad \Longleftrightarrow \models \mathtt{self.next} \doteq \mathtt{null} \lor (\models \mathtt{self.next} \not\doteq \mathtt{null}\\
&\qquad\quad \land \models \mathtt{self.i\ >\ self.next.i} \land \models \mathrm{inv}_{\mathtt{Node}}(\mathtt{self.next}))
\end{aligned}$$

Additionally to this recursive predicate, we add to Γ the fact that any defined non-null object will satisfy this invariant:

$$\forall\ \mathtt{self}.\ \ \models \partial\ \mathtt{self} \land \models \mathtt{self} \not\doteq \mathtt{null} \longrightarrow\ \models \mathrm{inv}_{\mathtt{Node}}(\mathtt{self})$$

Our recursive definitions are a conjunction of the explicit invariant and the multiplicity constraints of our example; we used Δ-short form in order not to clutter up our presentation too much, i. e., facts like $\models \partial$ `self.i` were omitted. The invariant $\mathrm{inv}_{\mathrm{Node}}$ `@pre` expresses well-formedness in a pre-state:

$$\begin{aligned}
&\forall\ \mathtt{self}.\ \ \models \partial\ \mathtt{self} \land \models \mathtt{self} \not\doteq \mathtt{null}\\
&\qquad \longrightarrow\ \models \mathrm{inv}_{\mathtt{Node}}\mathtt{@pre(self)} \Longleftrightarrow \models \mathtt{self.next@pre} \doteq \mathtt{null}\\
&\quad \lor (\models \mathtt{self.next@pre} \not\doteq \mathtt{null} \land \models \mathrm{inv}_{\mathtt{Node}}\mathtt{@pre(self.next@pre)}\\
&\qquad\qquad \land \models \mathtt{self.i@pre\ >\ self.next@pre.i@pre})
\end{aligned}$$

3.3 Translating Contracts into HOL

Given the fact that $\models (\mathtt{true})_{pre}$ just collapses to `true`, the formulae that we add to Γ is the straight-forward simplification of the semantics rule Fact 2:

$$\begin{aligned}
&\forall\ \mathtt{self}.\ \Delta(\mathtt{self}) \longrightarrow \models \mathtt{self.contents()} \triangleq\\
&\qquad\qquad\quad \mathtt{if\ self.next} \doteq \mathtt{null\ then\ Set\{i\}}\\
&\qquad\qquad\quad \mathtt{else\ self.next.contents()\text{-}{>}including(i)}\\
&\qquad \land \neg\Delta(\mathtt{self}) \longrightarrow \models \mathtt{self.contents()} \triangleq \bot
\end{aligned}$$

where Δ`(self)` is a short-cut for $\models \partial$ `self` $\land \models$ `self` $\not\doteq$ `null`. The variant for `contents@pre()` looks as follows:

$$\begin{aligned}
&\forall\ \mathtt{self}.\ \Delta(\mathtt{self}) \longrightarrow \models \mathtt{self.contents@pre()} \triangleq\\
&\qquad\quad \mathtt{if\ self.next@pre} \doteq \mathtt{null\ then\ Set\{i\}}\\
&\qquad\quad \mathtt{else\ self.next@pre.contents@pre()\text{-}{>}including(i)}\\
&\qquad \land \neg\Delta(\mathtt{self}) \longrightarrow \models \mathtt{self.contents@pre()} \triangleq \bot
\end{aligned}$$

4 Test Generation

In the specification-based testing, we are interested in testing the formula ϕ—called *test specification*—in the statement:

$$\Gamma \vdash \phi$$

instead of proving it (the $\vdash$ is interpreted as implication). We are interested in test specifications which contain calls to operations s`.op(`a_1`,...,`a_n`)`; the core of the technique consists in selecting arguments consistent with the specification Γ

and the semantic rules for the operations of OCL, executing the implementation of `op` and checking if the result validates ϕ. We follow the classical approach of transforming the test specification into a disjunctive normal form (DNF), extended by invariant-handling and the treatment of recursive definitions, which corresponds to partitioning the input space of the operation(s).

Another class of case distinctions arises from *aliasing*; i. e., the fact that two object references can designate the same object, i. e., s`.next.next` is in fact identical to s due to a cycle in the object graph. Aliasing is a crucial phenomenon in object-oriented systems. It is likely that a system behaves differently depending on the aliasing relationships among the objects it handles. Therefore we will add further case distinctions to the specification under analysis that distinguish different aliasing relationships. We will refer to this transformation as *alias closure.*

4.1 Test Specifications: Getting Started

Depending on the specific test purposes, there are various ways to test a system: a test could be concerned with the normal behavior of operations, which will be the default considered here, or with exceptional behavior (what happens if the precondition is not satisfied?), or with operation sequences, e. g., $(\sigma, \sigma') \models \phi \land (\sigma', \sigma'') \models \psi$. It is even conceivable to express in test specifications the sharing of pre-state and post-state or different parameters; in our specification, the implementation of `insert` can have a copying semantics (all object-contents in the list were copied to freshly generated objects) as well as a sharing semantics.

Since OCL expressions cannot have side-effects, properties of non-query operations like `insert` cannot be expressed inside OCL. We therefore suggest to present the test specification directly in HOL. Thus, following Fact 2, we have for the case of a "normal behavior unit test":

```
  Δ(s,x) ⟶ ⊨ s.contents() ≐ s.contents@pre()->including(x)
∧ ¬Δ(s,x) ⟶ ⊨ s.insert(x) ≜ ⊥
```

where `s` is a free variable for which we look for solutions that meet all possible constraints (arising from the context Γ, but also locally in ϕ). Since $(\Delta \longrightarrow A) \land (\neg\Delta \longrightarrow B)$ is equivalent to $(\Delta \land A) \lor (\neg\Delta \land B)$ and the latter is closer to a DNF, we rewrite our test specification and have:

```
  Δ(s,x) ∧ ⊨ s.contents() ≐ s.contents@pre()->including(x)
∨ ¬Δ(s,x) ∧ ⊨ s.insert(x) ≜ ⊥
```

which boils down to:

```
  Δ(s,x) ∧ ⊨ s.contents() ≐ s.contents@pre()->including(x)
∨ ⊨ s ≐ null ∧ ⊨ s.insert(x) ≜ ⊥
∨ ⊨ x ≐ ⊥ ∧ ⊨ s.insert(x) ≜ ⊥
```

Here, we can already apply unit propagation in clauses and extract the two test cases: $\models$ `null.insert(x)` $\triangleq \perp$ and $\models$ `s.insert(`$\perp$`)` $\triangleq \perp$ which essentially test the corner-cases imposed by the semantics of OCL.

4.2 Test Hypotheses

The test cases $\models$ `null.insert(x)` $\triangleq \bot$ and $\models$ `s.insert(`$\bot$`)` $\triangleq \bot$ give also some deeper insight into testing. Test cases are *classes* of concrete tests, i. e., the ground instances (e. g., $\models$ `null.insert(0)` $\triangleq \bot$ or $\models$ `null.insert(1)` $\triangleq \bot$) of these formulae . Overall, a *test case* for an operation *op* in a DNF is a conjoint:

$$\models \phi_1(x_1, \dots, x_n) \wedge \dots \wedge \models \phi_n(x_1, \dots, x_n)$$

where at least one ϕ_i depends on *op*. In test cases, we can partition the clauses into two groups: in *oracles* O, i. e., those $\models \phi_i(x_1, \dots, x_n)$ that depend on *op*, and in *constraints* C, i. e., all others. Constructing a test boils down to finding a solution, i. e., a ground substitution for $x_1, \dots, x_n$, that satisfies all constraints in C, while the test proceeds by executing the implementation of *op* for this solution and check if the oracles evaluate to true.

Logically, this means that we made the assumption "if there is an input vector $(x_1, \dots, x_n)$ satisfying all constraints, and if this input passes the oracle execution, the oracles will pass for all inputs satisfying the constraints." This type of assumption underlying a test is called a *uniformity hypothesis* written:

$$\begin{aligned}&(\exists x_1, \dots, x_n.\ C(x_1, \dots, x_n) \wedge O(x_1, \dots, x_n))\\ &\qquad\longrightarrow (\forall x_1, \dots, x_n.\ C(x_1, \dots, x_n) \longrightarrow O(x_1, \dots, x_n))\end{aligned}$$

While this is the most fundamental testing hypothesis, there are other useful ones that help to establish case distinctions used in specification-based tests. A notable other well-known form of a testing hypothesis is the *regularity hypothesis*:

$$\begin{aligned}&(\forall x_1, \dots, x_n. |x_1, \dots, x_n| < k \wedge\ C(x_1, \dots, x_n) \longrightarrow O(x_1, \dots, x_n))\\ &\qquad\longrightarrow (\forall x_1, \dots, x_n.\ C(x_1, \dots, x_n) \longrightarrow O(x_1, \dots, x_n))\end{aligned}$$

or in other words: whenever we tested all data up to a given complexity measure (like size of collections) k, we assume that the execution of *op* satisfies the specification. In the context of OCL testing, there is a similar form of regularity hypothesis: here, we will implicitly argue over the bound k on the number of different objects in a state that has been used for the tests.

4.3 Unfolding

For generating a set of test cases, we start with the test specification given above, restricted to the part where `s` is defined and not `null`.

```
Δ(s,x) ∧ ⊨ s.contents() ≐ s.contents@pre()->including(x)
```

This test specification does not show any explicit case distinctions. Rather, the case distinctions are hidden in the recursive specification of `contents()`.

The invariants over the different arguments of the operation (including `s`) must be taken into account for the generation of relevant test cases. In our example, only ordered lists can occur in pre-states and post-states of the `insert` operation. Adding these invariants as constraints over the pre-states or post-states reduces the number of test cases derived from the test specification by removing as many non-satisfiable clauses as possible before the test data selection. Because of the facts contained in Γ, we obtain:

```
∀ self. ⊨∂ self ∧ ⊨ self ≠̇ null ⟶ ⊨ inv_Node(self)
```

These invariants can be inserted at any time during the unfolding process.

For instance, we can already insert the invariant for the pre-states and post-states of the `insert` operation, knowing that `s` is defined and not `null`:

```
Δ(s,x) ∧ ⊨ inv_Node@pre(s) ∧ ⊨ inv_Node(s)
∧ ⊨ s.contents() ≐ s.contents@pre()->including(x)
```

To enrich this condition with explicit case distinctions, we unfold the operation calls and invariants by replacing them with their specification: an operation call will be replaced with its contract and an invariant with its definition, which is allowed here since we have $\Delta(s, x)$. For the sake of readability, we do not replace the `contents` operation calls directly with their contract but rather conjoin the contract with the existing formulae. We obtain the following conditions:

```
  Δ(s,x)
∧ (⊨ s.next@pre ≐ null
  ∨ (⊨ s.next@pre ≠̇ null
   ∧ ⊨ s.i@pre > s.next@pre.i@pre ∧ ⊨ inv_Node@pre(s.next@pre)))
∧ (⊨ s.next ≐ null
  ∨ (⊨ s.next ≠̇ null ∧ ⊨ s.i > s.next.i ∧ ⊨ inv_Node(s.next)))
∧ ⊨ s.contents() ≐ s.contents@pre()->including(x)
∧ ⊨ s.contents() ≜ if s.next ≐ null then Set{s.i}
                    else s.next.contents()->including(s.i)
∧ ⊨ s.contents@pre() ≜
    if s.next@pre ≐ null then Set{s.i@pre}
      else s.next@pre.contents@pre()->including(s.i@pre)
```

A second refinement step could be performed by unfolding the invariants and the operation calls a second time: we could insert the invariant definitions again and instantiate the operation contract for the `contents` operation with `s.next` (correspondingly for the pre-state).

The unfolding process and invariant insertion can be stopped at any time, once the refinement is sufficient according to the tester's needs. Then, the DNF of the obtained formula is generated to enumerate the different test cases coming from case distinction. The DNF obtained for the previous formula is the following, leading to four clauses distinguishing whether `s.next` and `s.next@pre` are `null`.

```
 (Δ(s,x)
 ∧ ⊨ s.next ≐ null
 ∧ ⊨ s.next@pre ≐ null
 ∧ ⊨ s.contents() ≐ s.contents@pre()->including(x)
 ∧ ⊨ s.contents() ≜ Set{s.i}
 ∧ ⊨ s.contents@pre() ≜ Set{s.i@pre})
```

$$\begin{array}{l}
\lor\ (\Delta(s,x)\\
\quad \land \models \texttt{s.next} \not\doteq \texttt{null} \land \models \texttt{s.i > s.next.i} \land \models \mathrm{inv}_{\texttt{Node}}(\texttt{s.next})\\
\quad \land \models \texttt{s.next@pre} \doteq \texttt{null}\\
\quad \land \models \texttt{s.contents()} \doteq \texttt{s.contents@pre()->including(x)}\\
\quad \land \models \texttt{s.contents()} \triangleq \texttt{s.next.contents()->including(s.i)}\\
\quad \land \models \texttt{s.contents@pre()} \triangleq \texttt{Set\{s.i@pre\})}\\
\lor\ (\Delta(s,x)\\
\quad \land \models \texttt{s.next} \doteq \texttt{null}\\
\quad \land \models \texttt{s.next@pre} \not\doteq \texttt{null} \land \models \texttt{s.i@pre > s.next@pre.i@pre}\\
\quad \land \models \mathrm{inv}_{\texttt{Node}}\texttt{@pre(s.next@pre)}\\
\quad \land \models \texttt{s.contents()} \doteq \texttt{s.contents@pre()->including(x)}\\
\quad \land \models \texttt{s.contents()} \triangleq \texttt{Set\{s.i\}}\\
\quad \land \models \texttt{s.contents@pre()} \triangleq \texttt{s.next@pre.contents@pre()}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{->including(s.i@pre))}\\
\lor\ (\Delta(s,x)\\
\quad \land \models \texttt{s.next} \not\doteq \texttt{null} \land \models \texttt{s.i > s.next.i} \land \models \mathrm{inv}_{\texttt{Node}}(\texttt{s.next})\\
\quad \land \models \texttt{s.next@pre} \not\doteq \texttt{null} \land \models \texttt{s.i@pre > s.next@pre.i@pre}\\
\quad \land \models \mathrm{inv}_{\texttt{Node}}\texttt{@pre(s.next@pre)}\\
\quad \land \models \texttt{s.contents()} \doteq \texttt{s.contents@pre()->including(x)}\\
\quad \land \models \texttt{s.contents()} \triangleq \texttt{s.next.contents()->including(s.i)}\\
\quad \land \models \texttt{s.contents@pre()} \triangleq \texttt{s.next@pre.contents@pre()}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{->including(s.i@pre))}
\end{array}$$

The first case boils down (due to constant propagation and set reasoning) to:

$$\begin{array}{l}
\quad \Delta(s,x) \land \models \texttt{s.next} \doteq \texttt{null} \land \models \texttt{s.next@pre} \doteq \texttt{null}\\
\land \models \texttt{s.i} \triangleq \texttt{s.i@pre} \land \models \texttt{s.i} \triangleq \texttt{x}
\end{array}$$

All other cases are not yet "ground" enough and contain application redexes like $\models \mathrm{inv}_{\texttt{Node}}(\texttt{s.next})$ for further invariant unfolding. The derivation

$$\begin{array}{l}
\quad \Delta(s,x)\\
\land \models \texttt{s.next} \not\doteq \texttt{null} \land \models \texttt{s.i > s.next.i} \land \models \mathrm{inv}_{\texttt{Node}}(\texttt{s.next})\\
\land \models \texttt{s.next@pre} \doteq \texttt{null}\\
\land \models \texttt{s.next.contents()->including(s.i)} \doteq \texttt{Set\{s.i@pre\}}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{->including(x)}
\end{array}$$

for the second case expands to:

$$\begin{array}{l}
(\Delta(s,x)\\
\land \models \texttt{s.next} \not\doteq \texttt{null} \land \models \texttt{s.i > s.next.i} \land \models \texttt{s.next.next} \doteq \texttt{null}\\
\land \models \texttt{s.next@pre} \doteq \texttt{null}\\
\land \models \texttt{s.next.contents()->including(s.i)} \doteq \texttt{Set\{s.i@pre\}}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{->including(x))}\\
\lor\ (\Delta(s,x)\\
\land \models \texttt{s.next} \not\doteq \texttt{null} \land \models \texttt{s.i > s.next.i}\\
\land \models \texttt{s.next.i > s.next.next.i} \land \models \texttt{s.next.next} \not\doteq \texttt{null}\\
\land \models \mathrm{inv}_{\texttt{Node}}(\texttt{s.next.next})\\
\land \models \texttt{s.next@pre} \doteq \texttt{null}\\
\land \models \texttt{s.next.contents()->including(s.i)} \doteq \texttt{Set\{s.i@pre\}}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{->including(x))}
\end{array}$$

While the second sub-case is unsatisfiable since it asserts that the insertion increases the list length by two, the first sub-case reduces to:

$$\begin{aligned}
&\ \ \Delta(s,x)\\
&\land \models \texttt{s.next} \not\doteq \texttt{null} \land \models \texttt{s.i > s.next.i} \land \models \texttt{s.next.next} \doteq \texttt{null}\\
&\land \models \texttt{s.next@pre} \doteq \texttt{null}\\
&\land \models \texttt{Set\{s.next.i\}->including(s.i)} \doteq \texttt{Set\{s.i@pre\}}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\texttt{->including(x)}
\end{aligned}$$

which, due to set reasoning, corresponds to a test case in which the inserted element x is not already in the list. The test cases still containing an occurrence of the invariance predicate correspond to the class of "yet to be tested" test cases.

4.4 Alias Closure

Unfolding and invariant insertion represent only a first step of the exploration of the specification by case distinction. There is another implicit case distinction that needs to be considered, since the two references s and s.next could actually refer to the same object, due to a cycle in the object graph. We should then distinguish the cases where $\texttt{s.next} \triangleq \texttt{s}$ and where $\texttt{s.next} \not\triangleq \texttt{s}$. In the same way, we should distinguish the cases $\texttt{s.next@pre} \triangleq \texttt{s}$ and $\texttt{s.next@pre} \not\triangleq \texttt{s}$.

To handle these four cases in the test case generation, we add the following tautology, called *alias distinction*, to the unfolding of our test specification:

$$\begin{aligned}
&\ \ (\models \texttt{s.next} \triangleq \texttt{s} \lor \models \texttt{s.next} \not\triangleq \texttt{s})\\
&\land (\models \texttt{s.next@pre} \triangleq \texttt{s} \lor \models \texttt{s.next@pre} \not\triangleq \texttt{s})
\end{aligned}$$

In the cases $\texttt{s.next} \triangleq \texttt{s}$ and $\texttt{s.next@pre} \triangleq \texttt{s}$, the invariants evaluate to false due to the strict inequality, thus only the cases $\texttt{s.next} \not\triangleq \texttt{s}$ and $\texttt{s.next@pre} \not\triangleq \texttt{s}$ remain. Computing the DNF in our example leads to almost the same formula as in the previous subsection, where $\models \texttt{s.next} \not\triangleq \texttt{s} \land \models \texttt{s.next@pre} \not\triangleq \texttt{s}$ is added to each conjoint.

In the general case, the alias closure of a formula is the conjoint of the tautologies $\texttt{p} \triangleq \texttt{q} \lor \texttt{p} \not\triangleq \texttt{q}$ for all the references p and q occurring in the formula (all other reference pairs are not relevant for case-splitting; so when we decided to unfold the invariants to a certain depth, we also made a decision on the maximum path-sizes and finally the maximum number of nodes in a state). Formally, let Path(φ) be the set of path-expressions (references) occurring in a formula φ. We define AliasClosure(φ) as the set of formulae

$$\{\ \texttt{p} \triangleq \texttt{q} \lor \texttt{p} \not\triangleq \texttt{q} \mid \texttt{p}, \texttt{q} \in \mathrm{Path}(\varphi) \land \texttt{p}\ \text{non-identical to}\ \texttt{q}\ \}$$

This produces all possible objects graphs, instead of only tree-like structures.

4.5 Generating Test Object-Graphs from Test Cases

Finally, ground instantiations of the underlying object model (i. e., in our example, instances of singly-linked lists) need to generated. For example, a concrete state-pair $\tau = (\sigma, \sigma')$ that can be given for test case

Table 1. Sample set of resulting test cases

List in pre-state	Inserted element	List in post-state
$\boxed{3}$	3	$\boxed{3}$
$\boxed{5}$	9	$\boxed{9} \rightarrow \boxed{5}$
$\boxed{6} \rightarrow \boxed{1}$	1	$\boxed{6} \rightarrow \boxed{1}$
$\boxed{6} \rightarrow \boxed{3}$	5	$\boxed{6} \rightarrow \boxed{5} \rightarrow \boxed{3}$
$\boxed{8} \rightarrow \boxed{5} \rightarrow \boxed{1}$	5	$\boxed{8} \rightarrow \boxed{5} \rightarrow \boxed{1}$
$\boxed{7} \rightarrow \boxed{6} \rightarrow \boxed{5}$	9	$\boxed{9} \rightarrow \boxed{7} \rightarrow \boxed{6} \rightarrow \boxed{5}$

```
Δ(s,x) ∧ ⊨ s.next ≠̇ null ∧ ⊨ s.i > s.next.i
∧ ⊨ s.next.next ≐ null ∧ ⊨ s.next@pre ≐ null
∧ ⊨ Set{s.next.i}->including(s.i) ≐ Set{s.i@pre}
                                        ->including(x)
```

and the ground instance `s.insert(2)` of the operation call `insert()` is:

```
σ  = { oid₀ ↦ (|i = 3, next = null |)}
σ' = { oid₀ ↦ (|i = 3, next = oid₁|), oid₁ ↦ (|i = 2, next = null|)}
```

which describes the requirement that inserting 2 into the list that only contains the element 3 should result in the sorted singly-linked list that contains the elements 3 and 2. Table 1 shows a sample set of test cases resulting from an unfolding of the test specification up to a list length of 3. For every list length there are two test cases: one for the case that the inserted element is already in the list and one for the case of an actual insertion.

5 Integrating the Technique in HOL-TestGen

HOL-TestGen [10] is a specification and test case generation environment extending the interactive theorem prover Isabelle/HOL [18]. The HOL-TestGen method is two-staged: first, the original formula is partitioned into test cases by transformation into a normal form called test theorem. Second, the test cases are analyzed for ground instances (the test data) satisfying the constraints of the test cases. Particular emphasis is put on the control of explicit test hypotheses. Finally, HOL-TestGen supports the generation of test drivers that allow for validating that an implementation fulfills its abstract specification. As such, developing UML/OCL support for HOL-TestGen, including its integration into a formal model-driven development toolchain, e. g., [6], enables the validation that an implementation fulfills the test specifications given in UML/OCL.

Extending HOL-TestGen with support for OCL creates certain challenges: the unfolding of OCL invariants introduces a new kind of splitting rule that needs to be supported efficiently by the splitter algorithm of HOL-TestGen. Moreover, HOL-TestGen does not yet support conjoint clauses that have no reference to the program under test (resulting in a failure during test data form computation). This is motivated by the fact that we cannot generate test drivers for such conjoint clauses. In the future, the generated test drivers could either silently drop such test cases or, following the OCL semantics, test for a deadlock.

6 Related and Future Work

6.1 Related Work

While there are several works that discuss specification-based test case generation based on UML/OCL models, none of them supports the three-valuedness of OCL. The most closely related works [1, 2, 4, 24] are all inspired by the seminal work of Dick and Faivre [13] and, thus, share the idea of using symbolic DNF computation for partitioning the input space. Moreover, there are works using sequence diagrams as an input for test case generation, e. g., [16], or pairwise testing of OCL contracts, e. g., [19]. Finally, Gogolla et al. [14] apply random-testing strategies for analyzing properties of OCL specifications.

For program-based tests, there are two test data generators that apply symbolic techniques: Korat [5] and Java Pathfinder [23]. Korat [5] generates from preconditions and a bound on the number of nodes of data structures, an input partitioning by a combination of symbolic execution and (simple) constraint solving. The idea of integrating a symbolic state deeply inside the execution environment, i. e., inside a Java virtual machine (JVM) as suggested in JPF-SE [3] (a successor of Java Pathfinder [23]), substantially improved the approach and inspired systems such as Pex [21] (a model-based testing tools for the .net).

6.2 Future Work

HOL-TestGen's generation strategies are geared towards inductively generated data (such as enumeration types, or lists and sets). In this paper, we have shown how *co*-inductively generated data such as object graphs can be tackled. The described translation is a pre-computation step, but it remains to provide new tactical infra-structure to implement the unfolding strategies and the alias closure. The concrete model-generation for the resulting specification is a standard-game for SMT-based model-construction generators in HOL-TestGen.

References

[1] van Aertryck, L., Jensen, T.: UML-CASTING: Test synthesis from UML models using constraint resolution. In: Jézéquel, J.M. (ed.) AFADL'2003.

[2] Aichernig, B.K., Pari Salas, P.A.: Test case generation by OCL mutation and constraint solving. In: QSIC '05, pp. 64–71. IEEE Computer Society (2005).

[3] Anand, S., Pasareanu, C.S., Visser, W.: JPF-SE: a symbolic execution extension to Java PathFinder. In: Grumberg, O., Huth, M. (eds.) TACAS, *LNCS*, vol. 4424, pp. 134–138. Springer (2007).

[4] Benattou, M., Bruel, J.M., Hameurlain, N.: Generating test data from OCL specication. In: WITUML (2002)
[5] Boyapati, C., Khurshid, S., Marinov, D.: Korat: automated testing based on Java predicates. In: ISSTA, pp. 123–133 (2002).
[6] Brucker, A.D., Doser, J., Wolff, B.: An MDA framework supporting OCL. Electronic Communications of the EASST **5** (2006).
[7] Brucker, A.D., Krieger, M.P., Wolff, B.: Extending OCL with null-references. In: Gosh, S. (ed.) Models in Software Engineering, no. 6002 in LNCS, pp. 261–275. Springer (2009).
[8] Brucker, A.D., Wolff, B.: HOL-OCL: A Formal Proof Environment for UML/OCL. In: Fiadeiro, J., Inverardi, P. (eds.) FASE, no. 4961 in LNCS, pp. 97–100. Springer (2008).
[9] Brucker, A.D., Wolff, B.: An extensible encoding of object-oriented data models in HOL. Journal of Automated Reasoning **41**, 219–249 (2008).
[10] Brucker, A.D., Wolff, B.: HOL-TestGen: an interactive test-case generation framework. In: Chechik, M., Wirsing, M. (eds.) FASE, no. 5503 in LNCS, pp. 417–420. Springer (2009).
[11] Brucker, A.D., Wolff, B.: Semantics, calculi, and analysis for object-oriented specifications. Acta Informatica **46**(4), 255–284 (2009).
[12] Church, A.: A formulation of the simple theory of types. Journal of Symbolic Logic **5**(2), 56–68 (1940)
[13] Dick, J., Faivre, A.: Automating the generation and sequencing of test cases from model-based specifications. In: Woodcock, J., Larsen, P. (eds.) Formal Methods Europe, *LNCS*, vol. 670, pp. 268–284. Springer (1993)
[14] Gogolla, M., Hamann, L., Kuhlmann, M.: Proving and visualizing OCL invariant independence by automatically generated test cases. In: Fraser, G., Gargantini, A. (eds.) TAP, *LNCS*, vol. 6143, pp. 38–54. Springer (2010).
[15] Gogolla, M., Richters, M.: Expressing UML class diagrams properties with OCL. In: Clark, T., Warmer, J. (eds.) Object Modeling with the OCL, *LNCS*, vol. 2263, pp. 85–114. Springer (2002)
[16] Li, B.L., shu Li, Z., Qing, L., Chen, Y.H.: Test case automate generation from UML sequence diagram and OCL expression. In: Computational Intelligence and Security, pp. 1048–1052. IEEE Computer Society (2007).
[17] de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS, *LNCS*, vol. 4963, pp. 337–340. Springer (2008).
[18] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL—A Proof Assistant for Higher-Order Logic, *LNCS*, vol. 2283. Springer (2002).
[19] Noikajana, S., Suwannasart, T.: An improved test case generation method for Web service testing from WSDL-S and OCL with pair-wise testing technique. pp. 115–123. IEEE Computer Society (2009).
[20] Object Management Group: UML 2.2 OCL specification (2010). Available as OMG document formal/2010-02-01
[21] Tillmann, N., de Halleux, J.: Pex—white box test generation for .NET. In: Beckert, B., Hähnle, R. (eds.) TAP, *LNCS*, vol. 4966, pp. 134–153. Springer (2008).
[22] Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Grumberg, O., Huth, M. (eds.) TACAS, *LNCS*, vol. 4424, pp. 632–647. Springer (2007).
[23] Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng. **10**(2), 203–232 (2003).
[24] Weissleder, S., Schlingloff, B.H.: Quality of automatically generated test cases based on OCL expressions. In: ICST, pp. 517–520. IEEE Computer Society (2008).