What Good Are Strong Specifications?

Nadia Polikarpova* Carlo A. Furia* Yu Pei* Yi Wei* Bertrand Meyer*,†
*Chair of Software Engineering, ETH Zurich, Switzerland †ITMO National Research University, St. Petersburg, Russia
firstname.lastname@inf.ethz.ch

***Abstract*—Experience with lightweight formal methods suggests that programmers are willing to write specification if it brings tangible benefits to their usual development activities. This paper considers *stronger* specifications and studies whether they can be deployed as an incremental practice that brings additional benefits without being unacceptably expensive. We introduce a methodology that extends Design by Contract to write strong specifications of functional properties in the form of preconditions, postconditions, and invariants. The methodology aims at being palatable to developers who are not fluent in formal techniques but are comfortable with writing simple specifications. We evaluate the cost and the benefits of using strong specifications by applying the methodology to testing data structure implementations written in Eiffel and C#. In our extensive experiments, testing against strong specifications detects twice as many bugs as standard contracts, with a reasonable overhead in terms of annotation burden and run-time performance while testing. In the wide spectrum of formal techniques for software quality, testing against strong specifications lies in a "sweet spot" with a favorable benefit to effort ratio.**

I. Introduction

Many years of progress in the theory and practice of formal methods notwithstanding, writing software specifications[1] still seems to be "disliked by almost everyone" [1]. In many cases, this disliking is a consequence of a high cost/benefit ratio—perceived or real—of writing and maintaining accurate specifications on top of the code. After all, developers *will* write specifications as long as they are simple, have a straightforward connection with the implementation, and help them write and debug code better and faster. One example is Design by Contract [2], [3] where simple executable specifications, written in the same syntax as programming language expressions, support design, incremental development, and testing and debugging. Another one is test-driven development [4], where rigorously defined test cases play the role of specifications in defining correct and incorrect behavior. Experiences with these techniques show that providing lightweight specifications is an accepted practice when it brings tangible benefits and integrates well with the overall development process.

But what about *strong* specifications, which attempt to capture the entire (functional) behavior of the software? Should we dismiss them on the grounds that the effort required to write them is not justified against the benefits they bring in the majority of mundane software projects? This paper studies the impact of deploying strong behavioral specifications, in the form of *contracts* (pre- and postconditions and class invariants), for detecting errors in software using automatic testing.

[1]In this paper, we target *formal* specifications of *functional* properties.

Using strong contracts involves costs and possible benefits. Among the former we have the programming effort necessary to write such strong specifications and the runtime overhead of checking them during execution. The benefits may include finding more errors, finding more subtle errors, finding errors more quickly, and exposing errors in ways that are easier to understand and correct. Our **contributions** address the cost factors—by measuring and trying to mitigate them—and assess the benefits:

- Sect. III presents a methodology to write strong specifications—extending our previous work [5]—that does not require fluency in formal techniques because it is an extension of such traditional practices as Design by Contract. This is instrumental in reducing the programming effort associated with strong specifications.
- The methodology comes with tool support and specification libraries, so that strong specifications are usable with standard debugging and testing tools.
- Sect. IV and V describe an extensive empirical study that evaluates the use of strong contracts for real software and measures their costs and benefits in terms of defect detection.

The bulk of our empirical study targets EiffelBase, a library of generic containers and data structures (such as lists, tables, and trees), which has been in use in the Eiffel community for more than 20 years. The production version of EiffelBase includes simple contracts, a form of partial specification, that are nonetheless quite effective at finding implementation bugs automatically using contract-based random testing [6], where executable contracts serve as oracles and enable a push-button testing process. In the present paper, we augment the simple contracts that come with EiffelBase using the methodology discussed in Sect. III. The result is EiffelBase+: a version of EiffelBase with identical implementation but strong (mostly complete) specifications.

In an extensive set of experiments, we compare the effectiveness of random testing on EiffelBase and EiffelBase+, with the goal of assessing whether the additional effort invested into the strong contracts pays off in terms of quantity and complexity of the bugs found. Our **experiments show** that these measures dramatically increase when deploying strong specifications: random testing found twice as many bugs in EiffelBase+, and the simple contracts of EiffelBase would have uncovered none of the new bugs. The overhead size of specifications, in contrast, remains moderate, with the specification-to-code ratio going from 0.2 to 0.46.

978-1-4673-3076-3/13 © 2013 IEEE

ICSE 2013, San Francisco, CA, USA

```
merge_right (other: LINKED_LIST [G])
  require
    not after
    other ≠ Void
    other ≠ Current
  ensure
    count = old count + old other.count
    index = old index
  end
```

Fig. 1. Standard specification of routine *merge_right* in *LINKED_LIST*.

```
merge_right (other: LINKED_LIST [G])
  require
    -- As in Fig. 1
  modify sequence
  ensure
    sequence = old (sequence.front (index) +
                    other.sequence + sequence.tail (index + 1))
  end
```

Fig. 2. Model-based specification of routine *merge_right* in *LINKED_LIST*.

Our approach to writing strong specifications that are effective for testing is not limited to Eiffel programs. In a companion set of experiments, we applied the same technique to writing strong specifications for the DSA C# library [7]. and tested the result using Pex [8]; in this case too we discovered new bugs with reasonable additional effort.

II. Strong Specifications: An Example

The following example illustrates and justifies the use of strong specifications. Consider the EiffelBase class *LINKED_LIST*—Eiffel's standard implementation of linked lists. Like many containers in EiffelBase, *LINKED_LIST* includes an internal cursor to iterate over elements of the list. The query[2] *index* gives the cursor's position, which can be on any element of the list in positions 1 through *count*, or take the special boundary values 0 ("*before*" the list) and *count* + 1 ("*after*" the list). The attribute *count* denotes the number of elements in the list.

Fig. 1 shows the EiffelBase specification of *LINKED_LIST*'s routine (method) *merge_right*. The routine inserts another list *other* passed as argument into the current list (denoted **Current** in Eiffel, corresponding to **this** in Java and C#) immediately after the cursor position. For example, if **Current** stores the sequence of elements b·a·r·t with cursor positioned on the "r" (*index* = 3) and *other* stores o·n·e, *merge_right* changes **Current** to b·a·r·o·n·e·t. The precondition (**require**) specifies that the routine cannot be called when the cursor is *after*: there is no valid position to the right of it. It also demands that *other* be non-**Void** (**null** in Java and C#) and not aliased with the **Current** list: otherwise, merging is not well defined. The postcondition (**ensure**) describes some expected effects of executing *merge_right*: the **Current** list will contain as many elements as it contained before the call to *merge_right* (denoted by **old** *count*) plus the number of elements of the *other* list; and the cursor's position *index* will not change.

The contracts in Fig. 1 are a good example of the kind of specification that Eiffel programmers normally write [9]: it is correct and nontrivial, and it can help detect errors in the implementation, such as performing partial merges or incorrectly leaving the cursor at a different position. Unfortunately the specification is also incomplete, because it does not precisely describe the expected state of the list after merging. In fact, the current implementation of *merge_right* contains an error that is undetectable against the specification of Fig. 1. The error occurs in the special case of calling *merge_right* with cursor *before* the list (*index* = 0): the implementation will insert *other* at the second rather than at the first position. For example, merging f·o·l·d and u·n when the cursor is *before* yields f·u·n·o·l·d instead of the correct u·n·f·o·l·d.

Sect. III presents a methodology to write, with moderate effort, strong specifications that extend and, whenever possible, complete this kind of partial specification. Fig. 2 shows the strong specification obtained by applying the methodology to *merge_right*, the way it appears in EiffelBase+. As is common in most Eiffel projects, the programmer who wrote *merge_right* did a good job with the precondition, which is sufficiently detailed and need not be strengthened. The postcondition, however, turns into a single assertion that defines the *sequence* of elements stored in the list after calling *merge_right* as the concatenation (operator +) of three segments: **Current**'s original sequence up until position *index* (written *sequence.front* (*index*)), followed by *other*'s element sequence, followed by the original sequence from position *index* + 1 (written *sequence.tail* (*index* + 1)). This postcondition relies on an *abstract model* of the linked list in the form of a mathematical sequence of elements, which was already implicitly present above, in the informal description of the semantics of *merge_right*. Models blend well with Eiffel's standard specification constructs to help formalize programmers' intuitive understanding of data structures semantics. Using the strong postcondition in Fig. 2, completely automatic testing with the AutoTest tool [6] detected the error that occurs in *merge_right* when the cursor is *before*.

The postcondition in Fig. 2 describes how the sequence changes, but it does not say what does *not change*. Including the assertion *index* = **old** *index* from the original postcondition is not sufficient, as it only mentions one piece of state that does not change. Instead we include the assertion **modify** *sequence*, which means that *merge_right* may only modify the sequence of elements in the **Current** list and *nothing else*. Together pre-, postcondition, and modify clause give a complete specification of *merge_right* behavior, against which we can automatically test any implementation for correctness.

III. How to Write Strong Specifications

Writing good specification is hard; at least this is the common belief. Experience with Design by Contract suggests that programmers can competently write simple specifications if they can be expressed using familiar syntax. See for example the specification in Fig. 1, which refers to regular class queries such as *count* and *index*, also used in the implementation.

[2] A *query* is an attribute or a function [2].

Without further guidance and language support, however, programmers tend to write only partial specifications, because expressing complex properties is cumbersome. This section describes *model-based contracts* (MBC): a methodology to write strong specifications that structures and extends traditional Design by Contract. MBC includes simple guidelines to define the abstract model of a class (Sect. III-A), and to write pre- and postconditions of routines (Sect. III-B and III-C) and other, more advanced, specification elements (Sect. III-D and III-E).

The MBC approach supports writing strong specifications in a number of ways: *models* facilitate choosing the right level of abstraction and expressing complex behavioral properties concisely, while the structured discipline for writing postconditions and invariants, together with the notion of *completeness* (Sect. III-D), provides precise guidelines as to which properties are worth documenting in a contract, and when a contract is strong enough. While fostering rigor and accuracy in specifications, MBC is still palatable to practitioners because its notation is part of the programming language. When developing specifications for testing, as opposed to formal verification, MBC can be exploited *incrementally*: developers may skip writing the most advanced specification elements (for example, complex class invariants) while still getting strong specifications that are useful to detect subtle errors.

The following subsections present MBC using examples from EiffelBase. The few additional constructs introduced by MBC are highlighted in a different color and underlined in the examples (e.g., **modify**). The current presentation of MBC derives from previous work of ours [5], which focused on using strong specifications when designing new software. In this paper we adapt the principles introduced in [5] to the goal of supplying *existing software* with flexible strong specifications for runtime checking and automatic testing (see Sect. III-F). We also extend the specification methodology with new construct that handle framing (Sect. III-D) and complex class invariants (Sect. III-E).

A. Abstract Class Models

Writing strong specifications becomes simpler if we can readily express the *abstract state space* of classes and how it changes. Therefore, the first step in specifying a class with MBC is defining a *model* for the class: a set of mathematical elements that capture the abstract state space.

Syntactically, the annotation **model** (see Fig. 3) declares the abstract model of a class as a list of attributes or functions called *model queries*; each element listed after **model** is either a query of basic type (Boolean, integer, or object reference) already used in the implementation, or a *specification query*, meaning a query introduced solely to define the model. As part of our work on MBC, we developed the Mathematical Model Library (MML), a collection of immutable Eiffel classes that represent mathematical concepts useful for specification: sets, bags, sequences, maps, and relations. Specification queries make use of MML classes to represent complex components of class models. For example, *LINKED_LIST*'s model in Fig. 3

```
class LINKED_LIST [G]

  model sequence, index

  sequence: MML_SEQUENCE [G]
    status specification
    -- Specification query: sequence of elements in the list.

  index: INTEGER
    -- Internal cursor position.

  off: BOOLEAN
      -- Is the cursor not on a list element?
    ensure
      Result = not sequence.domain.has (index)
    end

invariant
  -- Model constraint
  0 ≤ index and index ≤ sequence.count + 1
  -- Internal representation constraint
  not sequence.is_empty implies last_cell.item = sequence.last
end
```

Fig. 3. Excerpt of *LINKED_LIST*'s MBC specification in EiffelBase+.

has two components: a specification function *sequence* with return type *MML_SEQUENCE* that gives the abstract sequence of elements stored in the list, and the ordinary class attribute *index* of integer type.

Class models should be expressive enough to formalize the class behavior as seen at the API level, without exposing implementation-specific details. In practice, it is usually easy to devise a model for a data structure using MML abstractions. Even for classes representing complex real-world concepts, such as an ATM or a flight scheduler, MML remains applicable if used incrementally to define partial yet useful behavioral properties.

B. Preconditions

The *precondition* of a routine defines when a call to the routine is valid. In practice preconditions appear to be the most widely and accurately used form of contract [9]. Therefore, MBC does not introduce special guidelines for writing preconditions.

C. Postconditions

The *postcondition* of a routine r describes the intended effects of executing r on the object state; it is a *relation* between the state just before (denoted using the keyword **old**) and the state just after executing r.

MBC postconditions express the intended effect of executing a routine on the *model*, that is in terms of the model queries. Procedure *merge_right* in Fig. 2, for example, declares its effect on the model query *sequence* of the current object. For *functions*, the postcondition also mentions the returned object (and its model queries) using the keyword **Result**. For example, function *off* in Fig. 3 defines **Result** in terms of *sequence* and *index*.

D. Framing Specification

An accurate routine specification should limit the effects of the routine execution to a certain part of the program state. Such specification elements are called *framing specifications*.

In MBC, the keyword **modify** introduces a routine's framing specification: a list of all model queries whose value is allowed to change after executing the routine. For example, routine *merge_right* in Fig. 2 may only change *sequence*, but not *index* and not any component of the *other* list's model. The **modify** clause mechanism is taken from specification methodologies (e.g., Spec# [10]) targeted to formal correctness proofs; combined with models, the mechanism is also useful for finding bugs with testing.

This approach to framing also supports a simple definition of specification completeness: a routine postcondition and framing specification are *complete* if the relation between the model's pre- and poststate is a *function*.[3] Completeness is not an imperative in the MBC methodology: programmers can still approach writing postconditions and framing incrementally. It should rather be viewed as a safeguard against accidentally missing an important property.

E. Class Invariants

The *class invariant* specifies properties of valid instances of a class, which every operation must preserve. In MBC invariants, like postconditions, are expressed in terms of the model. For example, the first invariant clause in Fig. 3 constrains the values of the model queries *sequence* and *index*, stating that *index* must never take values outside the interval $[0..sequence.count + 1]$.

Additionally, class invariants can express internal representation constraints, which relate the values of model queries to the private attributes of the class. For example, the second invariant clause in Fig. 3 says that the private attribute *last_cell* stores the same value as *sequence*'s last element (whenever the sequence is not empty). Unlike other MBC specifications, invariants of this type do not describe the public interface of the class and usually cannot be made complete without revealing unnecessary implementation details in the model. However, even in this limited form, they turned out to be very effective at revealing errors that corrupt object's internal representation (see Sect. V-A).

Class invariant semantics. Since the semantics of class invariants can be subtle, MBC introduces additional dedicated constructs for complex invariant properties. We borrow some ideas from the existing techniques developed for formal correctness proofs (e.g., [10], among many); unlike these sophisticated techniques, MBC's solution for class invariants does not target comprehensiveness, but it is easy to deploy and sufficient in practice for finding errors by testing, while avoiding spurious invariant violations.

Eiffel checks class invariants at the beginning and at the end of every qualified[4] call on an object of the class. This rule prevents checking the invariant whenever routines of a class call one another within the boundaries of a single object, in order to accomplish a common task, as the object will normally be inconsistent ("open") until all operations are completed. However, when multiple objects in a complex object structure collaborate on the same task, and their invariants depend on the state of the whole object structure, this semantics may lead to spurious invariant violations.

Consider an example derived from real code in EiffelBase: a binary tree data structure, where each node has a link to its *parent* and *left* and *right* children. An invariant states that the **Current** node is its parent's left or right child:

```
parent ≠ Void implies (parent.left = Current or parent.right =
    Current)
```

Routine *prune_left* removes **Current**'s left child as follows:

```
old_left := left
left := Void
if old_left ≠ Void then old_left.set_parent (Void) end
```

When *old_left.set_parent* (**Void**) is called to remove the back-link from **Current**'s child, *old_left*'s class invariant is violated: its parent's *left* is already set to **Void**. This invariant violation is spurious because the consistency of the tree is going to be restored before the end of the call to *prune_left*; in fact, the very reason for calling *set_parent* is removing the inconsistency.

In MBC objects are implicitly equipped with a Boolean attribute *is_open* that is set to true at the entry to every public routine call on the object and restored to its previous value when the routine terminates; class invariants are checked only if *is_open* is false. In addition MBC provides the keyword **depend** to declare that an invariant clause depends on the state of an attribute, and hence it should be checked only if the object attached to the attribute is closed. Annotating the invariant in the example with **depend** *parent* removes the spurious invariant violation (*old_left.parent* is **Current**, which is open).

In the few cases when fine-grained control over the opening of objects is necessary, MBC provides the **open** clause to specify which arguments of a routine, in addition to the target, should be *explicitly* opened when the routine begins execution and closed again when the routine terminates.

As we discuss in Sect. IV, in EiffelBase+ we had to deploy explicit **depend** and **open** annotations only in a very few cases, limited to doubly-linked list nodes, and binary and n-ary trees.

F. Runtime Support for Strong Specifications

Model-based postconditions and invariants can be checked at runtime and used in testing out of the box: with the same tools and user experience as standard Eiffel contracts. Model queries introduced for specification purposes are implemented as regular functions that compute the abstract model value from the concrete object state, and thus do not require explicit initialization or updates. The specification classes we provide in MML are also regular Eiffel classes, implemented in a functional style. Even though this approach to implementation of model queries and model classes potentially incurs a high runtime overhead, the experiment results in Sect. V confirm that using MBC for contract-based testing is feasible.

[3] Such notion of completeness is of course relative to the model.

[4] A call *t.r* is *qualified* when the target *t* is an object other than **Current**.

Newly introduced specification constructs, such as **modify**, **depend** and **open**, do not have any effect in the standard Eiffel semantics: they are specified using **note** meta-annotations (similar to Javadoc or C#'s meta-data). We have developed a simple tool that rewrites these annotations into plain Eiffel; for example, **modify** clauses become explicit postconditions such as *item* = **old** *item*. The MBC methodology is conservative, in that the class semantics is still sound if we ignore the special annotations; ignoring **modify** clauses, for instance, yields weaker, yet correct, postconditions.

IV. Using Strong Specifications: Experiments

We performed an extensive experimental evaluation to assess the benefits of using strong specifications for finding errors in software.

A. Research Questions

The overall goal of this evaluation is assessing and comparing the advantages and the cost of deploying strong specifications in the form of model-based contracts (MBC, described in Sect. III) when applied to automatic contract-based testing of real software.

This materializes into the following research questions:

1) Are strong specifications effective for finding faults in software?
2) Do strong specifications find subtle and complex faults?
3) Do strong specifications find faults in little testing time?
4) What is the performance overhead of checking strong specifications at runtime?
5) What is the development effort required to provide strong specifications for existing software?

To answer these questions, we conducted two sets of experiments, targeting software written in Eiffel (Sect. IV-B) and C# (Sect. IV-C). In both cases, we selected an open-source library, specified it following the MBC methodology, and extensively tested it with a standard automatic testing tool. The rest of this section discusses the experiments; Sect. V presents the results.

B. Eiffel Experiments

The main experiments target EiffelBase (rev. 506)—Eiffel's standard base library—from which we selected 21 classes of varying size and complexity. Using the facilities of the EiffelStudio IDE, we built the *flat* version of each class, which is a self-contained implementation including all inherited members explicitly in the class text. This simplified the task of writing specifications without being distracted by EiffelBase's deep multiple inheritance hierarchy. For each of the 21 classes in their flat version, Tab. I lists the size (in LOC) and the number of public routines (PR), possibly also including helper classes directly used in the class implementation. Since different classes may share some parent or helper classes, the totals at the bottom of the table are in general less than the sum of the elements in each column.

Like most Eiffel software, EiffelBase comes with partial specification in the form of contracts: the 21 classes include 561 precondition clauses, 985 postcondition clauses, and 250 class invariant clauses. In EiffelBase+ we completely replaced EiffelBase's original postconditions and class invariants with model-based annotations, but we kept EiffelBase's preconditions (with a few exceptions discussed below)[5]. EiffelBase+'s strong specification includes 589 precondition clauses, 1066 postcondition clauses and 164 class invariant clauses, as well as 278 **modify**, 4 **depend** and 7 **open** clauses. Tab. I shows the size (in LOC and PR) of EiffelBase+, which also includes model definitions and implementations of the model queries necessary to write MBC.

Preconditions. In all but two EiffelBase+ classes we kept the same preconditions as in EiffelBase. Within the specific setup of our experiments, where we compare traditional contracts and strong contracts, it is important to have the same preconditions in the two artifacts under comparison. Preconditions define the valid calling contexts of routines (in particular, contract-based testing tools use them to select valid test cases). Changing preconditions would change the semantics of classes in a way similar to changing implementation: strengthening a precondition may reduce the number of faults detectable for the routine, since it would move obligations from the routine to its clients; weakening a precondition may increase the number of faults, since it would impose a heavier burden on its implementation. We treat preconditions as developers' design decisions, which we normally take at face value. This policy makes the experiments with EiffelBase and EiffelBase+ fully comparable.

The only exception occurred with four routines of class *BINARY_TREE* and eight routines of class *TWO_WAY_TREE* that insert new nodes into a tree. In these twelve cases, we strengthened the preconditions to disallow creating cycles among nodes in the tree. Without the strengthening, tree instances can be driven into inconsistent states with cycles where the whole specification of trees would be inapplicable. These changes in preconditions are conservative: the EiffelBase+ experiments using these stronger preconditions miss a few faults that are detected in EiffelBase, because the new preconditions rule out some previously valid failing test cases. Since these changes affect only a small fraction of all the experiments, the results with EiffelBase and EiffelBase+ remain comparable.

Specification correctness. To write correct strong contracts with MBC, we analyzed the original implementation, contracts, and comments in EiffelBase, and relied on our informal knowledge of the semantics of data structures and their implementation. To increase our confidence in the correctness of the new specification, we ran a series of short preliminary testing sessions with the goal of detecting inconsistencies and inaccuracies. All our changes were conservative, in that whenever a new contract forbade a behavior that was not clearly forbidden by the comments, standard contracts, or informal knowledge, we weakened the specification to allow the behavior. In all, we reached a high confidence that EiffelBase+'s specification

[5]All the code developed as part of the study, as well as descriptions of found faults are publicly available online [11].

TABLE I
EIFFEL CLASSES UNDER TEST AND RESULTS.

	EiffelBase							EiffelBase+					
Class	LOC	PR	TC	Spec	Inc	Real	New	LOC	PR	TC	Inc	Real	New
ARRAY	831	53	2.8	2	0	2	1	986	59	1.2	0	3	2
ARRAYED_LIST	1840	86	3.5	0	0	0	0	2037	92	1.7	0	1	1
ARRAYED_QUEUE	537	32	1.8	0	0	2	0	648	37	3.8	0	2	0
ARRAYED_SET	1960	49	5.8	3	1	8	0	2053	58	5.4	0	16	8
BINARY_TREE	1122	64	1.0	2	5	6	0	1366	70	1.1	0	16	10
BOUNDED_QUEUE	558	32	1.4	0	0	2	0	659	37	3.8	0	2	0
HASH_TABLE	1345	51	0.9	1	0	1	0	1626	63	0.9	0	2	1
HASH_TABLE_ITERATOR	217	15	0.4	0	0	0	0	248	15	0.5	0	0	0
INDEXABLE_ITERATOR	186	14	1.0	2	0	0	0	228	15	2.7	0	0	0
INTEGER_INTERVAL	519	42	4.3	1	1	0	0	637	45	0.9	0	3	3
LINKED_LIST	1759	69	2.0	0	0	2	0	1942	77	2.5	0	5	3
LINKED_LIST_ITERATOR	311	15	0.7	0	0	0	0	357	16	0.7	0	0	0
LINKED_SET	2128	83	5.4	5	2	7	0	2410	94	4.8	0	24	17
LINKED_SET_ITERATOR	311	15	0.7	0	0	0	0	357	16	0.7	0	0	0
LINKED_STACK	1077	27	1.0	0	0	3	1	1078	32	3.2	0	6	4
TWO_WAY_LIST	2007	71	0.8	0	0	3	0	2184	79	2.2	0	6	3
TWO_WAY_LIST_ITERATOR	412	15	0.7	0	0	0	0	462	16	0.7	0	0	0
TWO_WAY_SORTED_SET	2706	91	5.3	5	2	9	0	2983	102	4.8	1	34	25
TWO_WAY_SORTED_SET_ITERATOR	412	15	0.7	0	0	0	0	462	16	0.7	0	0	0
TWO_WAY_TREE	2548	90	1.4	4	4	22	5	2865	101	1.3	0	29	12
TWO_WAY_TREE_ITERATOR	412	15	0.7	0	0	0	0	462	16	0.7	0	0	0
Total	**17841**	**1033**	**42.5**	**15**	**12**	**48**	**7**	**19400**	**1164**	**44.4**	**1**	**103**	**62**

LOC: Lines of code, PR: Public routines, TC: Test cases drawn (million)
Spec: Specification errors found, Inc: Inconsistency errors found, Real: Real faults found, New: Faults found only in this experiment

is correct and strong enough. The results of the main testing sessions (Sect. V) corroborate this informal assessment.

Testing experiments. We ran a large number of random testing sessions with the AutoTest framework [6] on a computing cluster of the Swiss National Supercomputing Centre, configured to allocate a standard 1.6 GHz core and 4 GB memory to each parallel AutoTest session. The experiments totalled 1680 hours of testing time that generated nearly 87 millions of test cases; the TC columns in Tab. I list the million of test cases drawn when testing each class in EiffelBase and in EiffelBase+. The testing of every class was split into 30 sessions of 80 minutes, each with a new seed for the random number generator, such that corresponding sessions in EiffelBase and EiffelBase+ use the same seeds. This thorough testing protocol guaranteed statistically significant results [12].

C. C# Experiment

A smaller set of experiments targets 9 classes from DSA (v. 0.6)—an open-source data structure and algorithm library written in C# [7]. Support for contracts in C# appeared only recently, through the Code Contracts framework [13]; therefore, most C# projects (including DSA) do not have any formal specification. This was a chance to extend the validation of the MBC methodology to other languages and to projects without pre-existing specification.

We instructed one of our bachelor's students to follow the methodology of Sect. III and create DSA+: a variant of DSA with the same implementation but equipped with strong model-based contracts. DSA+'s specification includes 6 precondition clauses, 143 postcondition clauses and 23 class invariant clauses. For each of the 9 classes, Tab. II shows the size (in LOC and PR) of both DSA and DSA+, inclusive of all specification elements and model query implementations. As in Tab. I, the count also includes (possibly shared) helper classes. Flattening was not necessary in this case because the inheritance hierarchy is shallow.

TABLE II
C# CLASSES UNDER TEST AND RESULTS.

	DSA		DSA+		Testing	
Class	LOC	PR	LOC	PR	T	F
AvlTree	345	6	391	7	23	1
BinarySearchTree	205	5	213	5	21	1
CommonBinaryTree	419	13	536	18	83	0
Deque	201	14	231	15	145	0
DoublyLinkedList	408	17	458	19	171	3
Heap	371	11	390	12	61	1
OrderedSet	136	9	158	11	10	0
PriorityQueue	186	13	216	14	65	0
SinglyLinkedList	439	20	492	22	148	3
Total	**3043**	**133**	**3486**	**149**	**727**	**9**

LOC: Lines of code, PR: Public routines
T: Testing time (minutes), F: Faults found

Specification correctness. We manually inspected the DSA+ specification written by our student, and assessed its quality to be comparable to that of EiffelBase+ in terms of correctness and completeness. Since DSA was not designed with contracts in mind, it makes recurrent usage of defensive programming, throwing exceptions to signal invalid arguments. The experiment setup is consistent with this programming style: we do not consider such exceptions to be faults.

Testing experiments. We performed automatic testing with the Pex concolic testing framework [8] running on a Windows box equipped with a 2.16 GHz Intel Core2 processor and 3 GB of memory. The experiments ran for about 12 hours; column T in Tab. II reports the breakdown per class in minutes. The

testing time is different from class to class because Pex testing sessions by default are limited by coverage criteria rather than duration. We only tested DSA+ since DSA has no formal specification elements usable as automated testing oracles.

V. Using Strong Specifications: Results

This section discusses the result of the experiments, focusing on the larger EiffelBase experiments, with V-A through V-E targeting the research questions 1–5 of Sect. IV-A. Then, V-F briefly discusses the experiments with C#, and V-G presents possible threats to validity of the results.

A. Faults Found

AutoTest found 75 faults in EiffelBase and 104 in EiffelBase+; these are *unique*, that is they identify distinct and independent errors. We classified them in three categories.

Specification faults correspond to violations of *wrong* contracts (meaning that in our judgement they specify the expected behavior of the program incorrectly). We found 15 specification faults in EiffelBase (column Spec in Tab. I) and none in EiffelBase+, which increased our confidence that the preliminary testing sessions mentioned in Sect. IV-B were sufficient to achieve correct specifications. We consider specification faults spurious in our study, because we are not comparing the correctness of the specification in EiffelBase and EiffelBase+ but rather their effectiveness at finding real errors in the implementation.

Inconsistency faults correspond to failures triggered by calls on objects in inconsistent states, which are not captured by a partial class invariant. For example, *LINKED_SET* may be driven into a state where the container stores duplicate elements; calling *remove* (x) in such a state triggers a failure (only one occurrence of x is removed), but *remove* is not to blame for it, since it is due to previous erroneous behavior that went undetected. While inconsistency faults are genuine errors, we classify them separately because understanding and locating the ultimate source of an inconsistency is normally harder. Additionally, a single inconsistency fault often results in many failing test cases (potentially in all routines of the class that rely on the broken invariant), requiring additional effort from the developer when analyzing the testing results.

We found 12 inconsistency faults in EiffelBase and 1 in EiffelBase+ (columns Inc in Tab. I); the ultimate source of the latter fault is a class invariant not including all internal representation constraints (see Sect. III-E), which would have required exposing implementation details in the model. The other inconsistency faults of EiffelBase are not detected in EiffelBase+, because, due to stronger class invariants, their *real* source is detected instead. In the *LINKED_SET* example above, instead of the inconsistency fault in *remove*, MBC report a fault in routine *replace*, which does not check if the new value is already present in the set, thereby introducing duplicates. The results in this category indicate that strong specifications report faults in a way that is easier to understand and debug.

All other errors are *real faults* which correspond to genuine errors directly traceable to the code. We found 48 real faults in EiffelBase and 103 in EiffelBase+ (columns Real in Tab. I); 41 of them are found in both sets of experiments, 7 only in EiffelBase, and 62 only in EiffelBase+. We submitted bug reports for all the 110 faults found in our experiments. The Eiffel Software developers in charge confirmed 107 (97%) of them as real bugs to be fixed. This is evidence that we are dealing with genuine faults in our evaluation. The remaining three faults not taken on by the developers also arguably highlight real problems in the implementation, but they are probably not so likely to occur during "normal" runs. The rest of the discussion focuses on real faults unless stated otherwise.

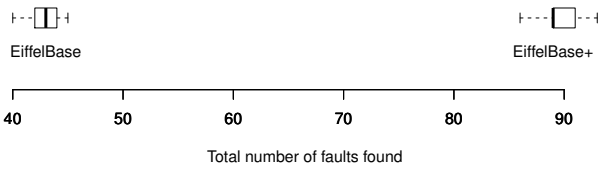

Fig. 4. Unique real faults found in all classes over 80-minute testing sessions.

Only seven faults are found in EiffelBase but not in EiffelBase+ (columns New in Tab. I). Four of them are prevented by the strengthened preconditions in the tree classes (Sect. IV-B); two are shadowed by new failures occurring earlier; and one disappears with MBC due to an unintentional side-effect of a model query that amends an invariant violation. None of these faults found only in EiffelBase show inherent deficiencies of strong specifications or of the MBC method. In contrast, the 62 faults found only in EiffelBase+ are undetectable in EiffelBase.

Except for the two *ITERATOR* classes (no faults in both cases) and the two *QUEUE* classes (the same two faults in both cases), the number of faults found is consistently higher in EiffelBase+ *in each class*. As evident from the boxplot in Fig. 4, the difference is highly significant: the Mann-Whitney U test gives $U = 0$ (testing EiffelBase+ outperforms testing EiffelBase in *all* sessions), and $p = 2 \cdot 10^{-11}$ overall and $p \leq 2.1 \cdot 10^{-11}$ for every class (except the *ITERATOR*s and *QUEUE*s). The difference remains highly statistically significant even if we aggregate the experiments in sessions of different length.

> Testing with strong specifications detected 55 more (twice as many) unique real faults than testing with standard, partial contracts. 62 (56%) of the faults are detected *only* with strong specifications.

B. Fault Complexity

Although it is to some extent subjective whether a fault is "deep" or "subtle", faults violating postconditions or class invariants are arguably more complex because so are the violated properties. While there is no significant difference in the percentage of class invariant violations between EiffelBase and EiffelBase+ (33% in both cases), postconditions trigger 42% of violations in EiffelBase+ but only 11% in EiffelBase:

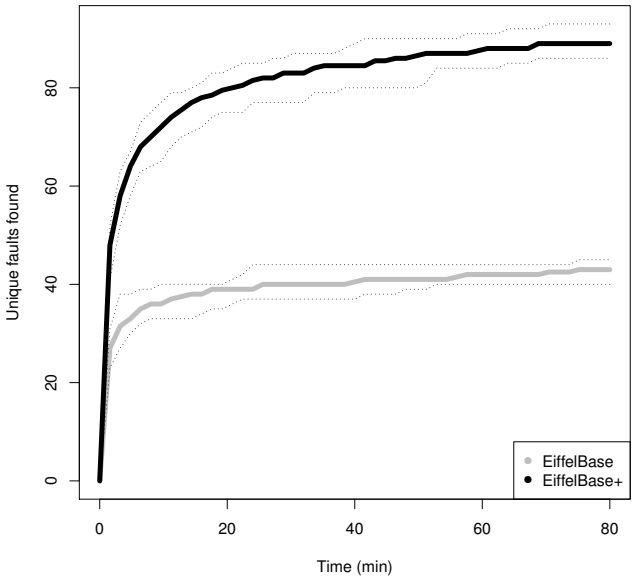

Fig. 5. Median number of faults, aggregated from all classes, in time. Dotted lines show minimum and maximum for each case.

the Wilcoxon signed-rank test among all classes gives $W = 0$ and $p = 6 \cdot 10^{-3}$ both for postconditions alone and for postconditions and class invariants counted together, which demonstrates that strong specifications systematically detect more complex errors. 76% of faults in EiffelBase+ are detected thanks to postconditions or invariants—a direct consequence of the effectiveness of the MBC methodology for writing them.

We do not have enough space to present even a sample of the faults found only in EiffelBase+ and demonstrate that they are indeed subtle yet understandable (see the extended version of this report [14] and the online materials for details). It is revealing, however, that 11 faults in EiffelBase+ are due to violations of contracts generated automatically by our tool that processes MBC annotations (Sect. III-F) such as **modify** and **depend**. These faults are practically out of the scope of regular contracts, as specifying the corresponding properties explicitly is extremely onerous.

C. Usage of Testing Time

Fig. 5 plots the number of faults detected in EiffelBase and EiffelBase+ over a median 80-minute session; it is clear that the behavior with strong specifications dominates over standard contracts after only a few minutes. Dominance is observed consistently in all classes (with the usual exception of *ITERATOR*s and *QUEUE*s): a median session with strong contracts finds more faults than a median session with standard contracts after a time between two seconds and five minutes depending on the class under test; after a time between 13 seconds and 20 minutes, testing with strong contracts finds more faults than testing with standard contracts will find in the whole session.

TABLE III
SPECIFICATION OVERHEAD

# TOKENS	EIFFELBASE	EIFFELBASE+	OVERHEAD
Preconditions	1514	1696	1.12
Postconditions	5410	11837	2.19
Invariants	1508	1587	1.05
MBC annotations		1893	
Model queries		2268	
Total	**8432**	**19281**	**2.29**
Spec/code	0.20	0.46	

Testing with standard contracts also seems to exhaust earlier its fault-finding potential: given any time from 20 minutes on, there are more EiffelBase sessions than EiffelBase+ sessions that have found all the faults they ever will by this time. This may indicate that standard contracts are good to find "quick to detect" faults, but they also soon run out of steam.

We considered other differences between experiments with EiffelBase and with EiffelBase+ in the usage of testing time: repeatability of testing session history, and the presence of *rare* faults triggered only in a small number of cases. Our experiments with strong specifications are slightly less repeatable and include a few more rare faults, but the differences with standard contracts are not statistically significant.

D. Runtime Performance Overhead

Runtime checking of strong specifications based on models often requires traversing the whole data structure to construct an object of a model class, whenever a contract element is exercised. As a rule, this demands more computational resources than executing the simple checks involved in standard contracts. To measure the runtime overhead of checking MBC specifications in automated testing, we compared the number of test cases generated by AutoTest in the same amount of time when testing EiffelBase and EiffelBase+. Contrary to our expectations, the overhead is small in many cases and not significant overall (see column TC of Tab. I). A possible interpretation of this data is that the overhead of strong specifications grows as larger data structures are instantiated; because random testing most of the time only exercises small data structures, this overhead does not show.

We did not find a significant correlation between the variation of overhead for different classes and any source code metrics we considered. On the other hand, some AutoTest heuristics that decide to discard previously created objects are activated more often for classes where strong specifications are faster to check.

E. Specification Writing Overhead

Applying MBC to create EiffelBase+ required roughly one person-month, plus one person-week of preliminary testing for fine-tuning the specification, which puts the overall ratio benefit/effort at about four defects detected per person-day. Tab. III measures the amount of work produced in this time: for each specification item, including preconditions, postconditions, class invariants, MBC annotations such as **modify**, and model query implementations, we compare the number

of *tokens* in EiffelBase+ against those in EiffelBase (when applicable) and give the OVERHEAD of strong specifications as the ratio of the two values. The last line also shows the overall specification to code ratios.

Reflecting the importance MBC gives to strong postconditions and the more restricted role of class invariants, 67% of all *new* specification in EiffelBase+ are postconditions, whereas only 9% are class invariants. MBC-specific annotations are 11%, mostly **modify** clauses that are however straightforward to write and dispense for more intricate explicit framing specifications. Model query implementations account for the remaining 13%.

These numbers suggest that the specification overhead of MBC is moderate and abundantly paid off by the advantages in terms of errors found and quality of documentation. The specification to code ratio also compares favorably to other approaches to improving software quality. Detailed quantitative data about industrial experiences with test-driven development is scarce, but few references indicate [4], [15], [16] that it is common to have between 0.4 and 1.0 lines of tests per line of application code for projects of size comparable to EiffelBase. Correctness proofs are normally much more demanding, as they require between 1.5 and 9 specification elements per implementation element [17], [18], [19].

F. C# Experiments

Pex found 9 unique faults in DSA+ violating the model-based specification (column F in Tab. II). Unfortunately, we could not get an evaluation of these faults by the original code developers. We have confidence, however, that the faults uncover some obvious errors and, even in the most benign interpretation, some instances of bad object-oriented design.

The fault rates (faults per line of executable code) are comparable in the Eiffel and C# experiments, being respectively $6 \cdot 10^{-3}$ and $3 \cdot 10^{-3}$. The fault complexity is also qualitatively similar for the two languages. The testing time (column T in Tab. II) is instead incomparable, as Pex and AutoTest implement very different testing algorithms.

Applying MBC to create DSA+ required roughly 50 person-hours, plus another 8 person-hours used by the student to learn the MBC methodology on small examples. The specification/code ratio is perceptibly higher in DSA+ compared to EiffelBase+ (0.9); this is largely due to the verbose syntax of Code Contracts which are a library, as opposed to Eiffel's native language support for contracts.

G. Threats to Validity

Threats to internal validity of our findings come from the usage of randomized testing tools, whose behavior may change in different sessions. We designed the experimental protocol [12] to reduce this threat to a minimum: we ran a large number of repeated experiments and we performed suitable non-parametric statistical tests of significance for all differences we observed.

Threats to external validity refer to the generalizability of our findings. While MBC leads to very good results in our experiments, applying it to programs in application domain other than data structures might be more difficult or require an extension of the technique. Our results remain significant, however, if compared to the state of the art in deploying strong specifications. The generalizability to other languages and analysis tools is partially addressed by our experiments targeting two languages (Eiffel and C#) and two automatic testing technologies (random and concolic). Future work will experiment with even more approaches and notations.

VI. RELATED WORK

This section discusses the most significant related work in three areas: using formal specifications for testing; using inferred specifications to improve testing; and model-based specification methods.

Formal specifications for testing. The idea of using formal specifications for testing has a history that stretches back more than three decades; see [20] for a comprehensive survey. Various proposals targeted different specification formalisms including algebraic datatypes [21], [22], logic-based notations [23], UML Statecharts [24] and other state machines, and contracts and similar forms of embedded assertions [25], [26], [27], [6]. In these applications, formal specifications provide reliable—often automated—testing oracles [28] and can also guide test planning and test case generation.

This extensive experience is evidence that formal specifications can improve the testing process. From a software engineering viewpoint, however, an outstanding open issue is finding optimal trade-offs between the effort required to provide formal specifications and the improvements (in efficiency and effectiveness) they bring to the testing of real software. The evidence—empirical [29] or anecdotal [1]—is scarce in this area: most successful experiences do not explicitly take into account the effort required to produce reliable specifications against the benefits gained for testing (e.g., [30]); or they only target partial specifications, which have the advantage of being easy to write (e.g., [27], [6]). In contrast, this paper targeted the high-hanging fruit of deploying strong specifications, explicitly addressing the difficulties of writing and using such specifications for existing software. Our results that strong specifications reveal complex (design) errors corroborate Hoare's view that the real value of tests is that "they detect inadequacy in the [development] methods" [31].

Inferred specifications for testing. When specifications can be inferred automatically from the code, the deployment effort is negligible compared to the benefits they bring. Therefore, a number of recent works (e.g., [32], [33], [34], [35]) developed sophisticated techniques for inferring specifications from program executions with the intent of using them to improve testing. The experiments reported in these papers show that inferred specifications can boost automated testing [36]; on the other hand, even the most accurate inferred specifications only express the code from a different angle, and hence cannot take the developer's intent fully into account and are necessarily limited to detecting certain types of inconsistencies.

Combining inferred and manually written specifications is an interesting endeavor that belongs to future work (see [9], [37] for some preliminary studies).

Model-based specification methods. The methodology described in Sect. III extends our previous work [5] with the specific goal of developing executable specifications for automated testing. The same goal has also motivated the techniques to improve the runtime checking of strong specifications described in Sect. III-F. The related work section of [5] compares the foundations of our model-based method against other similar approaches such as JML [38].

VII. Conclusions and Future Work

This paper presents a methodology to write strong specifications that extends the traditional Design by Contract, and applied it to specifying data structure classes in Eiffel and C#. We carried out an extensive empirical evaluation to determine the benefits of using such strong specifications in testing with automatic tools. We found twice as many bugs in the software with strong specifications as in the same software specified with standard partial contracts. We also demonstrated that the effort required to write the strong specifications was moderate thanks to the methodology that is practical and palatable to professionals not fluent in formal techniques.

The benefits brought by strong specifications are not limited to finding errors through testing. While the present paper focused on adding strong specifications to existing code a posteriori, our related work [5] shows that model-based contracts help achieve consistent designs and higher-quality code by construction.

As **future work**, we plan to extend the MBC methodology and supporting tools to work on more complicated application domains with a higher degree of automation, and to support other software analysis techniques such as correctness proofs and static analysis. We will also expand the experimental evaluation to more projects and programming languages.

Acknowledgements

Thanks to Rosemary Monahan and Scott West for comments on a draft of this paper; and to Tobias Kiefer for carrying out the C# experiments. This work was partially supported by the Swiss SNF under projects FullContracts (200021-137931) and ASII (200021-134976); we also acknowledge the support of the Swiss National Supercomputing Centre for the testing experiments.

References

[1] D. L. Parnas, "Precise documentation: The key to better software," in *The Future of Software Engineering*. Springer, 2011, pp. 125–148.
[2] B. Meyer, *Object Oriented Software Construction*. Prentice Hall, 1997.
[3] P. Chalin, "Are practitioners writing contracts?" in *The RODIN Book*, ser. LNCS, vol. 4157, 2006, p. 100.
[4] K. Beck, *Test-Driven Development*. Addison-Wesley, 2002.
[5] N. Polikarpova, C. A. Furia, and B. Meyer, "Specifying reusable components," in *VSTTE*, ser. LNCS, vol. 6217, 2010, pp. 127–141.
[6] B. Meyer, A. Fiva, I. Ciupa, A. Leitner, Y. Wei, and E. Stapf, "Programs that test themselves," *IEEE Computer*, vol. 42, no. 9, pp. 46–55, 2009.
[7] http://dsa.codeplex.com/.
[8] N. Tillmann and J. de Halleux, "Pex–white box test generation for .NET," in *TAP*, 2008, pp. 134–153.
[9] N. Polikarpova, I. Ciupa, and B. Meyer, "A comparative study of programmer-written and automatically inferred contracts," in *ISSTA*, 2009, pp. 93–104.
[10] M. Barnett, K. R. M. Leino, and W. Schulte, "The Spec# programming system: an overview," in *CASSIS*. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 49–69.
[11] http://se.inf.ethz.ch/people/polikarpova/mbctesting.
[12] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *ICSE*. ACM, 2011, pp. 1–10.
[13] http://research.microsoft.com/en-us/projects/contracts/.
[14] N. Polikarpova, C. A. Furia, Y. Wei, Y. Pei, and B. Meyer, "What good are strong specifications?" Extended version available at http://arxiv.org/abs/1208.3337.
[15] N. Nagappan, E. M. Maximilien, T. Bhat, and L. Williams, "Realizing quality improvement through test driven development: results and experiences of four industrial teams," *ESE*, vol. 13, pp. 289–302, 2008.
[16] E. M. Maximilien and L. Williams, "Assessing test-driven development at IBM," in *ICSE*, 2003, pp. 564–569.
[17] V. Klebanov *et al.*, "The 1st verified software competition," in *FM*, ser. LNCS, vol. 6664, 2011, extended version at www.vscomp.org.
[18] F. Dupressoir, A. D. Gordon, J. Jürjens, and D. A. Naumann, "Guiding a general-purpose C verifier to prove cryptographic protocols," in *IEEE Computer Security Foundations Symposium*, 2011.
[19] J.-C. Filliâtre, "Verifying two lines of C with Why3: an exercise in program verification," in *VSTTE*, ser. LNCS, 2012, pp. 83–97.
[20] R. M. Hierons *et al.*, "Using formal specifications to support testing," *ACM Comput. Surv.*, vol. 41, no. 2, 2009.
[21] J. D. Gannon, P. R. McMullin, and R. G. Hamlet, "Data-abstraction implementation, specification, and testing," *ACM Trans. Program. Lang. Syst.*, vol. 3, no. 3, pp. 211–223, 1981.
[22] J. Chang and D. J. Richardson, "Structural specification-based testing: Automated support and experimental evaluation," in *ESEC/FSE*, 1999, pp. 285–302.
[23] P. Stocks and D. A. Carrington, "Test templates: A specification-based testing framework," in *ICSE*, 1993, pp. 405–414.
[24] A. J. Offutt and A. Abdurazik, "Generating tests from UML specifications," in *UML*, 1999, pp. 416–429.
[25] D. Marinov and S. Khurshid, "TestEra: A novel framework for automated testing of Java programs," in *ASE*, 2001, p. 22.
[26] Y. Cheon and G. T. Leavens, "A simple and practical approach to unit testing: The JML and JUnit way," in *ECOOP*, 2002, pp. 231–255.
[27] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: automated testing based on Java predicates," in *ISSTA*, 2002, pp. 123–133.
[28] M. Staats, M. W. Whalen, and M. P. E. Heimdahl, "Programs, tests, and oracles," in *ICSE*, 2011, pp. 391–400.
[29] M. M. Müller, R. Typke, and O. Hagner, "Two controlled experiments concerning the usefulness of assertions as a means for programming," in *ICSM*, 2002, pp. 84–92.
[30] L. du Bousquet, Y. Ledru, O. Maury, C. Oriat, and J.-L. Lanet, "Reusing a JML specification dedicated to verification for testing, and vice-versa: Case studies," *J. Autom. Reasoning*, vol. 45, no. 4, pp. 415–435, 2010.
[31] C. A. R. Hoare, "How did software get so reliable without proof?" in *FME*, 1996, pp. 1–17.
[32] C. Pacheco and M. D. Ernst, "Eclat: Automatic generation and classification of test inputs," in *ECOOP*, 2005, pp. 504–527.
[33] T. Xie and D. Notkin, "Tool-assisted unit-test generation and selection based on operational abstractions," *Autom. Softw. Eng.*, vol. 13, no. 3, pp. 345–371, 2006.
[34] A. Zeller, "Mining specifications: A roadmap," in *The Future of Software Engineering*. Springer, 2010, pp. 173–182.
[35] Y. Wei, H. Roth, C. A. Furia, Y. Pei, A. Horton, M. Steindorfer, M. Nordio, and B. Meyer, "Stateful testing: Finding more errors in code and contracts," in *ASE*, 2011, pp. 440–443.
[36] M. d'Amorim, C. Pacheco, T. Xie, D. Marinov, and M. D. Ernst, "An empirical comparison of automated generation and classification techniques for object-oriented unit testing," in *ASE*, 2006, pp. 59–68.
[37] Y. Wei, C. A. Furia, N. Kazmin, and B. Meyer, "Inferring better contracts," in *ICSE*, 2011, pp. 191–200.
[38] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok, "How the design of JML accommodates both runtime assertion checking and formal verification," *Sci. Com. Prg.*, vol. 55, no. 1-3, pp. 185–208, 2005.