Inferring Better Contracts

Yi Wei Carlo A. Furia Nikolay Kazmin Bertrand Meyer
Chair of Software Engineering, ETH Zürich, Switzerland
{yi.wei, carlo.furia, bertrand.meyer}@inf.ethz.ch nkazmin@student.ethz.ch

ABSTRACT

Considerable progress has been made towards automatic support for one of the principal techniques available to enhance program reliability: equipping programs with extensive contracts. The results of current contract inference tools are still often unsatisfactory in practice, especially for programmers who already apply some kind of basic Design by Contract discipline, since the inferred contracts tend to be simple assertions—the very ones that programmers find easy to write. We present new, completely automatic inference techniques and a supporting tool, which take advantage of the presence of simple programmer-written contracts in the code to infer sophisticated assertions, involving for example implication and universal quantification.

Applied to a production library of classes covering standard data structures such as linked lists, arrays, stacks, queues and hash tables, the tool is able, entirely automatically, to infer 75% of the complete contracts—contracts yielding the full formal specification of the classes—with very few redundant or irrelevant clauses.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software Verification—*programming by contract, assertion checkers, reliability*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

Keywords

contract inference, invariants, data mining, random testing

1. INTRODUCTION

Contracts are widely recognized as one of the most promising techniques for achieving software reliability. The advantages of equipping software with contracts include [22] not only avoiding mistakes in the first place and documenting the software accurately, but also, if mistakes do remain, improving dramatically the effectiveness of tests (and opening up the possibility of proofs, as well as automatic correction [32]). To yield the full range of these potential benefits, the contracts must be present throughout the software, and must describe its semantics as completely as possible. Most practical software, however, comes equipped with no contracts or, in languages enjoying support for Design by Contract [2], with incomplete contracts. To compensate, researchers have explored techniques (surveyed in Section 7) for inferring contracts from the code. Some of these techniques, such as abstract interpretation, involve static program analysis; others such as Daikon [10] are dynamic and rely on data from program executions.

These developments have yielded many important results and insights, but their practical usability remains limited. The advantages and limitations of the two kinds of techniques are complementary: Static approaches report sound but conservative contracts, and are mostly successful with small programs. Dynamic approaches scale better, but tend to be limited to inferring simple properties, such as a postcondition in a list insertion routine stating that "the list size has been increased by 1"; they also produce properties that are redundant with others or subsumed by them, such as "the list has grown in size" in this example, and unsound properties that characterize the test suite rather than the program. Even when relevant and non-redundant, the simpler properties typically inferred may be disappointing to programmers writing in a contract-equipped language, as they often are the kind of contract elements that programmers naturally intuit; what they expect from a supporting tool is that it will infer the more sophisticated properties.

The present work describes techniques for inferring advanced contracts, especially in the case of a contract-equipped language where the tools can take advantage of simple, partial contracts written by the programmers. Its focus is on *postconditions* of *commands* (routines changing the state of objects, as opposed to *queries*, which return information about an object); it more specifically looks for two important kinds of command postcondition clauses:

- Clauses involving quantification, to express for example that all previous elements of a structure are still present. Such clauses, which programmers seldom write, are critical to express frame properties.
- Clauses involving *implications* whose premises are single formulae or conjunctions of formulae.

Both kinds of postconditions are difficult to handle with most existing approaches without customized instrumentation. We demonstrate that when simple contracts are avail-

ICSE '11, May 21–28, 2011, Waikiki, Honolulu, HI, USA

Listing 1: Programmer-written and automatically inferred postconditions of *LINKED_LIST.extend*.

```
1  extend (v: G) -- Add 'v' to end. Do not move cursor.
2    ensure
3    -- Programmer-written
4    post_1: occurrences (v) = old (occurrences (v)) + 1
5
6    -- Automatically inferred
7    post_2: count = old count + 1
8    post_3: i_th (old count + 1) = v
9    post_4: forall i .
10     1 ≤ i ≤ old count implies i_th (i) = old i_th (i)
11   post_5: old after implies index = old index + 1
12   post_6: not old after implies index = old index
13
14   post_7: last = v
15   post_8: forall o: G ≠ v .
16     occurrences (o) = old occurrences (o)
17   post_9: forall o: G ≠ v . has (o) = old has (o)
```

able, as in contract-equipped languages, our techniques can infer stronger contracts fully automatically.

Experiments show that the proposed techniques, which we have implemented in the AutoInfer tool, inferred 75% of the complete postconditions for data structure classes such as linked list, queue, stack, array and hash table from EiffelBase, a production library which has been available for many years and is widely used in Eiffel applications. The inferred contracts achieve a high relevance, in that the various clauses are complementary and not mere repetitions, and accurately record the effect of each command on the visible state of the data structure.

AutoInfer is entirely automatic. It relies on dynamic analysis techniques and a general catalog of contract patterns. To construct the test suite that serves as a basis for contract inference, AutoInfer relies on the AutoTest [23] test generation framework.

Section 2 gives an example of the kind of sophisticated contracts that AutoInfer is able to infer. Section 3 describes the automatic generation of the test suite for classes under analysis. Section 4 explains the calculation of *change profiles*. Section 5 describes how to infer sophisticated postconditions involving functions with arguments, quantifications and implications. Section 6 evaluates the effectiveness of the proposed inference techniques. After the discussion of related work in Section 7, Section 8 presents future work. Section 9 draws conclusions.

2. EXAMPLE AND OVERVIEW

Class *LINKED_LIST* is the standard Eiffel implementation of linked lists, using dynamic allocation. A typical query is *occurrences* (*v*: *G*): *INTEGER*, returning the number of times its argument *v* of generic type *G* appears in the list. A typical command is *extend* (*v*: *G*), extending the list by adding *v* at the end.

The class authors equipped many features with contracts. For example, they annotated *extend* with the postcondition clause *post_1* in Listing 1, to state that the command increases the number of occurrences of the inserted element *v* by one. (The listing only shows the command header and the postcondition, ignoring the precondition, body, and other clauses).

As it is often the case with contracts written by programmers [25], the postcondition clause *post_1* is interesting but incomplete, as it fails to mention the command's effect on other components of the object state, described by queries other than *occurrences*. To be *complete* [26], the postcondition should also state that:

- The element *v* has been inserted at the end of the list.
- All the elements that were in the list before executing *extend* are still there, in the same positions.
- The internal "cursor" of the list does not change position.

The techniques described in the present paper, implemented in our AutoInfer tool, can automatically infer postconditions clauses *post_2* to *post_6* in Listing 1, which completely formalize the effects of *extend*:

- *post_2* asserts that exactly one element is added.
- *post_3* states that *v* appears at position **old** *count* + 1: the position next to the last valid position of the list before invoking *extend*.
- *post_4* states that for every valid position *i* of the **old** list, the element at position *i* is the same before and after the call.
- *post_5* and *post_6* formalize the position of the cursor in terms of its integer position *index*: if the *cursor* was *after* the last element of the list before invoking *extend*, it is now after the new last element, corresponding to an increment of *index* by one; otherwise *index* is unchanged.

Even though *post_2*-*post_6* capture every effect of *extend* observable through the queries of *LINKED_LIST*, redundant clauses may still be desirable to increase the readability and usefulness of the postcondition. AutoInfer indeed infers three additional clauses, listed as *post_7* to *post_9* in Listing 1: *v* is the *last* element of the list (*post_7*); the number of times every element other than *v* appears in the list is unchanged (*post_8*); any element other than *v* occurs in the list after insertion if and only if it occurred before (*post_9*).[1]

Overview. Figure 1 gives an overview of the inference techniques described in the rest of the paper, and of how AutoInfer integrates them.

1. The input is an Eiffel class, equipped with public queries, commands, and possibly a few basic contracts.
2. Using the random testing techniques implemented in AutoTest, AutoInfer generates a *test suite* that exercises as many routines as possible (Section 3).
3. AutoInfer executes the test suite and profiles the results, monitoring in particular which arguments and values returned by queries change when executing a command. This information is collected in a *change profile* (Section 4).

[1]In Listing 1, **forall** *o*: *G* ≠ *v* . ... is a shorthand for **forall** *o*: *G* . *o* ≠ *v* **implies** ... .

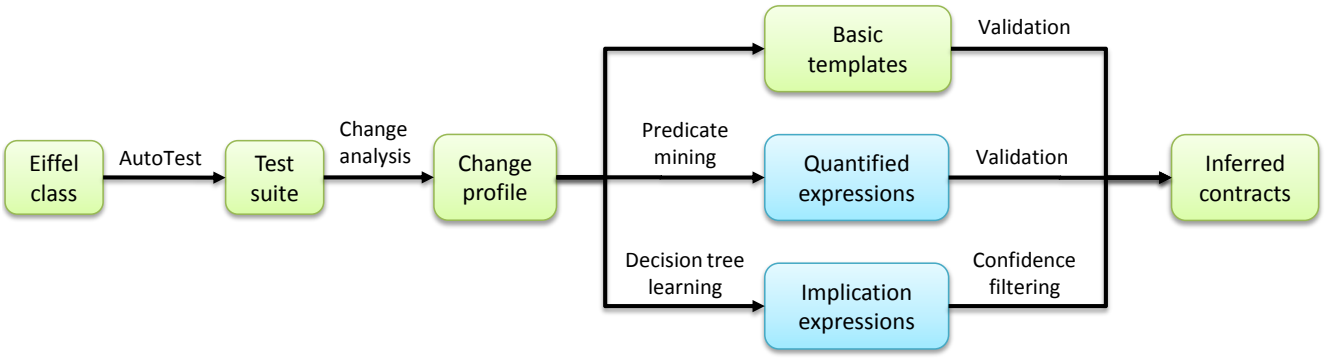

Figure 1: How AutoInfer works.

4. AutoInfer performs template-based invariant detection *à la* Daikon [10] to infer dynamically the simplest contract clauses. The information in the change profile is used to activate selectively a restricted number of templates, thus minimizing the inference of uninteresting or redundant clauses (Section 5.3). For example, *post_2* and *post_7* in Listing 1 are discovered using this technique.

5. AutoInfer mines the change profile for *predicates* appearing in expressions which evaluate to different values in the pre- and post-state of some execution. It then instantiates a number of pre-defined templates with the predicates to obtain a set of *quantified expressions*, such as *post_3*, *post_4*, *post_8*, and *post_9* in Listing 1. The expressions are candidate contracts, which are validated against the test suite: the candidates passing all test cases are likely valid contracts (Section 5.1).

6. AutoInfer also applies machine learning techniques, based on *decision tree learning* algorithms, to expressions in the change profile. The goal is finding correlations between expressions that change or stay constant. Every correlation with 100% confidence translates into a likely valid contract in the form of an implication (Section 5.2). For example, decision tree learning inferred *post_5* and *post_6* in Listing 1 by observing queries that change when executing *extend*.

7. Finally, AutoInfer collects all the inferred contracts and displays them to the user.

3. TEST SUITE GENERATION

AutoInfer starts from *dynamic* techniques. Dynamic techniques observe the program state over multiple executions at pre-defined points such as the entry and exit of a routine, and generalize these observations into likely valid contracts based on a given set of templates. The inferred contracts are only as good as the test suite which exercised the program; sound and interesting contracts require a set of test cases which execute program paths extensively and with varied values.

AutoInfer takes advantage of the existing AutoTest random testing framework [23] to build test suites completely automatically. AutoTest works on Eiffel classes equipped with contracts. Its test generation mechanism produces a pool of random objects by calling constructors and commands with random arguments. Then, to generate a test case for a routine r of a class C, it selects from the pool a target object of class C and a collection of objects and other values whose types make them appropriate as actual arguments for r. The precondition of r serves as a filter on the objects in the pool to select a valid input; the postcondition of r serves as an oracle to determine if the test case executed correctly or exposed a bug.

In this work we focus, as noted, on inferring postconditions of commands. The reason for this choice is the empirical observation that preconditions are usually much simpler than postconditions; in fact programmers often do write complete preconditions [25]. We modified AutoTest to discard failing test cases and increased its capability of exercising commands with the precondition satisfaction technique developed in recent work [31]. The test suites built with this method proved to be appropriate for dynamic contract inference techniques. Previous authors using dynamic techniques have stated that test suites produced with random-test generation are unsuitable for contract inference [10, 14]. Our own experience shows otherwise in the case of programs that are already equipped with a few simple contract elements, with the goal of inferring additional, more sophisticated assertions.

4. CHANGE PROFILE

AutoInfer runs the test suite generated by AutoTest, evaluates a set of expressions at the entry and exit points of a command, and determines how the command changes their values. The result is a *change profile* for the command, consisting of the values and their changes. The change profile is the basis for deciding which templates to activate during the next phase, contract inference (Section 5). The rest of this section explains how to select expressions and compute their change profiles.

The key goal, apparent in the notations introduced below, is to go beyond the simple properties found by most previous tools (such as *count* = **old** *count* + 1), involving queries with no arguments, and infer properties involving more complex queries with arguments, as in *occurrences* (*v*) = **old** *occurrences* (*v*) + 1.

4.1 Monitored entities

Whenever a test case exercises a routine, AutoInfer keeps track of the value of a number of entities before and after

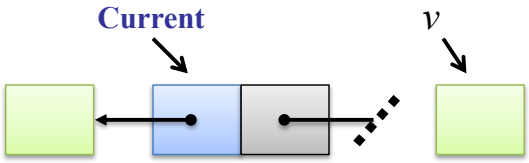

Figure 2: An instance of *LINKED_LIST*.

executing the routine's body.

The following notations are useful, for a command c of a class C with formal arguments $a_1, \ldots, a_m$ and an n-argument public query q of a class D:

- S^c denotes the set $\{\textbf{Current}, a_1, \ldots, a_m\}$ of *symbolic values* corresponding to the current object (*this*, *self*) and the formal arguments of c.
- $T^{q,c} \subseteq S^c$ denotes the set of symbolic values in S^c of type conforming to D: every element in $T^{q,c}$ is a type-correct *target* for calls to q.
- $A_i^{q,c} \subseteq S^c$ denotes the set of symbolic values in S^c of type conforming to the *i-th argument* of q.
- $E^{q,c}$ denotes the set

$$\left\{ a_0.q(a_1, \ldots, a_n) \mid a_0 \in T^{q,c} \wedge \bigwedge_{1 \le i \le n} a_i \in A_i^{q,c} \right\}$$

of type-correct *expression calls* to q with target and arguments from S^c.

Example. Consider the command *extend* (*v*: *G*) of *LINKED_LIST* and the query *has* (*v*: *G*) of the same class. Then, S^{extend} is $\{\textbf{Current}, v\}$; $T^{has,extend}$ is $\{\textbf{Current}\}$; $A_1^{has,extend}$ is $\{\textbf{Current}, v\}$, assuming that the generic type G allows nested lists; $E^{has,extend}$ is $\{\textbf{Current}.\mathit{has}\ (v),$ $\textbf{Current}.\mathit{has}\ (\textbf{Current})\}$.

4.2 Value sets

Consider an execution k of a command c. For every public query q, $V_k^{q,c}$ is the set of triples $\langle e, v, v' \rangle$ such that $e \in E^{q,c}$ is an expression call to q; v is the value returned by evaluating e right before the execution k of c; and v' is the value returned by evaluating e right after the same execution.

Example. Continuing the example of Section 4.1, consider an invocation k of command *extend* (*v*: *G*) on the list of Figure 2. $V_k^{has,extend}$ is then $\{\langle \textbf{Current}.\mathit{has}\ (v), \textbf{False}, \textbf{True} \rangle, \langle \textbf{Current}.\mathit{has}\ (\textbf{Current}), \textbf{False}, \textbf{False} \rangle\}$.

The *value, pre, post, and change sets* summarize the effects of a command c on the values of a public query q in every execution.

- The *value set* $\mathcal{V}_e^{q,c}$ of command c on *expression* $e \in E^{q,c}$ is the set

$$\left\{ \langle v, v' \rangle \mid \langle e, v, v' \rangle \in \bigcup_k V_k^{q,c} \right\}$$

of all values e evaluates to in any execution of c.

- The *pre set* $\overleftarrow{\mathcal{V}}_e^{q,c}$ of c on $e \in E^{q,c}$ is the set $\{v \mid \langle v, v' \rangle \in \mathcal{V}_e^{q,c}\}$ of all values e evaluates to before executions of c.
- The *post set* $\overrightarrow{\mathcal{V}}_e^{q,c}$ of c on $e \in E^{q,c}$ is the set $\{v' \mid \langle v, v' \rangle \in \mathcal{V}_e^{q,c}\}$ of all values e evaluates to after executions of c.
- The *change set* $\widetilde{\mathcal{V}}_e^{q,c}$ of c on $e \in E^{q,c}$, defined only for calls to queries returning numeric values, is the set $\{v' - v \mid \langle v, v' \rangle \in \mathcal{V}_e^{q,c}\}$ of all differences between the values e evaluates to after and before executions of c.

An expression $e \in E^{q,c}$ is *variant* with a command c iff $\mathcal{V}_e^{q,c}$ contains at least a pair $\langle v, v' \rangle$ such that $v' \neq v$. The notion of variant expression guides the selection of potential consequents in contracts involving implications (Section 5.2).

Examples. The following examples, again applying to *extend* of *LINKED_LIST*, illustrate the notions of pre, post, and change set.

- The query call **Current**.*is_empty* determines whether the current list is empty. *extend* may execute on empty or non-empty lists, always yielding a non-empty list. Hence, for every non-trivial test suite, the pre set of *extend* on **Current**.*is_empty* is $\{\textbf{True}, \textbf{False}\}$, the post set is $\{\textbf{False}\}$, and **Current**.*is_empty* is variant.
- The query call **Current**.*count* returns the number of elements in the list. *extend* always increases *count* by one, so the change set of *extend* on **Current**.*count* is $\{1\}$. The post set contains all positive integers from 1 to the maximum length of a list generated in the test suite.

5. INFERENCE TECHNIQUES

This section describes the inference techniques implemented in AutoInfer. 5.1 illustrates how to generate candidate contracts involving quantification, 5.2 discusses the generation of contracts with implications, 5.3 describes how the inference of basic contracts integrates with the other techniques, and 5.4 presents validation techniques for candidate contracts.

5.1 Contracts with quantification

Many interesting contracts compare the values returned by queries for different values of their arguments; such contracts can be formalized with expressions involving quantification. Quantified expressions are particularly useful in contracts of container classes: executing a command typically introduces *local* changes to the data structure, for example by adding or removing an element; a complete contract, however, should also mention explicitly that the state of the rest of the container has *not* changed. Such invariance properties are also called *frame properties*.

Consider, for example, the command *start* of the running example class *LINKED_LIST*, which moves the internal cursor to the first position in the list. While the effect of *start* on the cursor is easily described in a contract, expressing the absence of effects on the elements of the list requires contracts involving quantification such as **forall** *o* . *has* (*o*) = **old** *has* (*o*), which states that the list contains an object *o* after executing the command iff it contained it before.

Which query calls should appear in contracts involving quantifications? AutoInfer mines the change profile for calls to public queries on targets accessible from the current command. In principle, AutoInfer can consider queries with any

number of arguments. To simplify the presentation, however, we focus here on single-argument queries.

Consider the set of expressions defined by the grammar

$$\mathcal{P}^c \ni P \quad ::= \quad \textbf{True} \mid t.q \mid \textbf{not}\ P \mid \textbf{old}\ P$$

where q ranges over the set of all public queries and t ranges over $T^{q,c}$. For $p \in \mathcal{P}^c$, let $p(o)$ denote the call expression where the query appearing in p (if any) is applied to argument o. For each choice of expressions[2] $x, y, z \in \mathcal{P}^c$ whose arguments have compatible types, AutoInfer instantiates the following templates as *candidate postconditions* for c.[3]

$$\textbf{forall}\ o\ .\ x(o) = y(o)$$
$$\textbf{forall}\ o\ .\ x(o) \neq y(o)$$
$$\textbf{forall}\ o\ .\ x(o)\ \textbf{or}\ y(o)\ \textbf{implies}\ z(o)$$
$$\textbf{forall}\ o\ .\ x(o)\ \textbf{and}\ y(o)\ \textbf{implies}\ z(o)$$

For each $x, y, z \in \mathcal{P}^c$, if the command c has a formal argument v of type compatible with the type of o in $x(o), y(o)$, $z(o)$, AutoInfer also instantiates the following templates.

$$\textbf{forall}\ o \neq v\ .\ x(o) = y(o)$$
$$\textbf{forall}\ o \neq v\ .\ x(o) \neq y(o)$$
$$\textbf{forall}\ o \neq v\ .\ x(o)\ \textbf{or}\ y(o)\ \textbf{implies}\ z(o)$$
$$\textbf{forall}\ o \neq v\ .\ x(o)\ \textbf{and}\ y(o)\ \textbf{implies}\ z(o)$$

The idea behind this second set of templates is that, if a command c operates on an argument v of the same type as some query q, it is likely to affect the values returned by the query differently according to whether q is invoked with v or with a different value as argument. For example, this is the case of command *extend* in Listing 1, whose postcondition clauses *post_8* and *post_9* have been inferred according to the "conditional" quantified templates.

Sequence-based contracts. Many container data structures can enumerate the elements they contain in a linear sequence. This obviously holds not only of lists but also of more complex structures such as trees through standard traversal mechanisms. An indicator that a data structure may admit an enumeration of its elements is the presence of a public query with a single argument of type *INTEGER* and a precondition constraining the values that the argument can take to an interval of the integers. Whenever a query *en* (*i*: *INTEGER*) with these characteristics exists, we denote by σ_{en} the sequence of values *en*(*m*), *en*(*m*+1),..., *en*(*M*) it induces, where $[m..M]$ is the integer interval determined by *en*'s precondition [32]. For example, the query *i_th* of class *LINKED_LIST* returns a reference to the *i*-th element in the list, provided $1 \leq i \leq count$. This induces the sequence σ_{i_th} of values $i\_th(1), \ldots, i\_th(count)$.

Whenever a sequence representation σ_{en} is available, AutoInfer instantiates a number of candidate contracts that try to capture the effects of any command c on the sequence of values σ_{en}. These contracts are built by instantiating the template:

$$\sigma_{en} \ = \ (\textbf{old}\ \sigma_{en}[m..x]) \circ z \circ (\textbf{old}\ \sigma_{en}[y..M]) \qquad (1)$$

with values for x, y, z, where $\sigma[a..b]$ denotes the subsequence of σ corresponding to the interval $[a..b]$ and $\circ$ denotes concatenation. Our experience with model-based contracts [26] suggested the particular form of (1), which is only a readable shorthand for the three candidate contracts with quantification in Listing 2—where ν abbreviates the expression $m +$ **old** $(x - m)$; and if $z = \lambda$ (λ denotes the empty sequence) then *c_2* is undefined and $\delta_z = 0$, otherwise $\delta_z = 1$.

The method for picking values for x, y, z is, informally, to use for x and z any two references or queries of suitable type (integer for x and the same as *en* for z) that can be evaluated in c's postcondition, for y to pick $x + 1$ or $x + 2$. Formally, let $w : q$ denote that the types of expressions w, q are conforming. Then, z ranges over the set

$$\{\lambda\} \cup \{w \in S^c \mid w : en\} \ \cup \ \bigcup_{q:en} E^{q,c}\,,$$

$\hat{x}$ ranges over the set

$$\{m, M\} \cup \{v \in S^c \mid v : INTEGER\} \ \cup \ \bigcup_{q:INTEGER} E^{q,c}\,,$$

x ranges over the set $\{\hat{x}, \hat{x}+1, \hat{x}-1\}$, and y ranges over the set $\{x+1, x+2\}$.

In the running example of Listing 1, *post_3* and *post_4* correspond to the instantiation of the sequence-based template for query *i_th* and $m = 1$, $x = M = count$, $z = v$, $y = x + 1$. For these values, $m =$ **old** m, $\nu =$ **old** *count*, and the interval $[y..M]$ is empty, hence *c_1* corresponds to *post_4*, *c_2* to *post_3*, and *c_3* reduces to **True**.

5.2 Contracts with implications

The natural form of many postconditions is an implication

$$p_1 \wedge p_2 \wedge \cdots \wedge p_n \implies p_0 \qquad (2)$$

which links several predicates $p_1, \ldots, p_n$ holding before executing a command to another predicate p_0 holding afterward. $p_1, \ldots, p_n$ are called *antecedents* and p_0 is the *consequent*. Intuitively, each p_i is a Boolean expression that compares the value returned by some query to another value. For example, *post_5* in Listing 1 asserts that, whenever *extend* is invoked on a list where *after* is **True**, the difference *index* − **old** *index* will be 1 (i.e., *index* is increased by 1).

Dynamic inference techniques, consisting in generating many candidate contracts by instantiating a template and then validating them against the test suite, do not work well to infer implications: there are often many different Boolean expressions that can instantiate the antecedents and the consequent of an implication, hence the resulting number of candidate contracts easily grows beyond manageable limits.

In the present paper, we use data mining techniques to search efficiently for correlations among predicates which can be expressed as an implication (2). This approach does not restrict *a priori* the expressions that can compose the predicates but relies on data mining algorithms to select and combine relevant expressions among a large number of candidates.

Listing 2: Sequence-based contracts.

```
1 c_1: forall m ≤ j ≤ n.en (j) = (old en) (j − m + old m)
2 c_2: en (ν + 1) = z
3 c_3: forall ν + 1 + δz ≤ j ≤ ν + 1 + δz + old (M − y) .
4            en (j) = (old en) (j − ν − 1 − δz + old y)
```

[2]Obviously redundant or trivial expressions are immediately discarded.

[3]All quantifications are implicitly restricted to objects of conforming type.

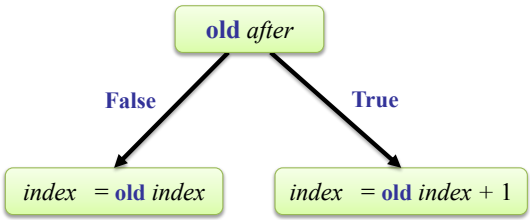

Figure 3: A decision tree for contracts of *extend*.

Decision tree learning. The approach uses *decision tree learning* [27]. The input to a decision tree learning algorithm consists of: (a) a set of *source* expressions $\{e_1, \ldots, e_n\}$, each with a finite set Δ_i of nominal values that it can take; (b) a *target* expression e_0 with its set Δ_0 of nominal values; (c) a training set $T \subseteq \Delta_1 \times \cdots \times \Delta_n \times \Delta_0$ of tuples of values for the source and target expressions. The output of the algorithm is a *tree* with a *confidence value* $0 \leq \alpha \leq 1$ where: (a) each leaf represents a value $\overline{e} \in \Delta_0$ of the target; (b) each interior node corresponds to one of the source expressions; (c) each node associated with some source e_i has at most $|\Delta_i|$ children, one for each value that e_i can take. The tree represents the target as a function of the sources that is consistent with a fraction α of the training set T: the value taken by e_0 for a source with values $\langle \overline{e}_1, \ldots, \overline{e}_n \rangle \in \Delta_1 \times \cdots \times \Delta_n$ is obtained by following the path that starts from the root and, for each node associated with expression e_i, continues to the child corresponding to value $\overline{e}_i$. For example, the simple decision tree in Figure 3 includes a single source Boolean expression *after* and an integer target expression *index* with nominal values $\{0, 1\}$; it represents the function $f : \{\textbf{True}, \textbf{False}\} \to \{0, 1\}$ defined as $f(\textbf{True}) = 1$, $f(\textbf{False}) = 0$.

We learn postconditions in the form of implications for a command c according to the following procedure.

1. Select any *target expression* t among all query expression calls from $\bigcup_q E^{q,c}$ that are *variant with* c.

2. Select a *nominal value set* Δ_t for t: if t is of type *INTEGER*, Δ_t can be the post set $\overrightarrow{\mathcal{V}}_t^{q,c}$ or the change set $\widetilde{\mathcal{V}}_t^{q,c}$; otherwise Δ_t is the post set $\overrightarrow{\mathcal{V}}_t^{q,c}$. When the change set is selected, we call t a *change target*.

3. The set of *source expressions* is $R_t \triangleq \bigcup_q E^{q,c} \setminus \{t\}$; each source expression $s_i \in R_t$ has nominal value set Δ_{s_i} equal to the pre set $\overleftarrow{\mathcal{V}}_{s_i}^{q,c}$.

4. Populate the training set T_t with a tuple $\langle \overline{s}_1, \ldots, \overline{s}_n, \overline{t} \rangle$ for each execution k of c. For each $1 \leq i \leq n$, $\overline{s}_i$ equals the value s_i evaluates to before the execution k of c. If t is a change target, then $\overline{t}$ is the difference between the values t evaluates to after and before the execution k of c; otherwise, $\overline{t}$ is the value t evaluates to after the execution k of c.

5. Feed the tree learning algorithm with the selected target t, source S_t, and training set T_t. AutoInfer uses RapidMiner [28], an open-source data mining toolkit, with multiple tree learning algorithms, and selects the decision tree with the highest confidence.

   The inference process is successful only if it builds a tree with confidence value 1. A decision tree with a lower confidence value disagrees with the values observed in some runs of the test suite, hence some of the implications it produces correspond to provably invalid contracts.

6. Rewrite each path[4] $e_1 \xrightarrow{\overline{e}_1} \cdots \to e_m \xrightarrow{\overline{e}_m} \overline{t}$ of the tree into the contract:

   $$\textbf{old}\ e_1 = \overline{e}_1 \ \wedge \cdots \wedge \textbf{old}\ e_m = \overline{e}_m\ \textbf{implies}\ \phi = \overline{t}$$

   where ϕ is t − **old** t if t is a change target and is just t otherwise. For example, under the assumption that *index* is a change target, the tree in Figure 3 corresponds to the contracts *post_5* and *post_6* in Listing 1.

Experiments suggested that decision tree learning is most likely to yield interesting implications when the target has a small nominal value set, typically with no more than 3 elements. The current implementation takes this fact into account to increase the efficiency of the inference process.

5.3 Basic contracts

In addition to the complex contract clauses generated with the techniques described in the rest of this section, AutoInfer implements some basic dynamic inference techniques to discover simpler contracts. These basic techniques generate candidate contracts by instantiating a pre-defined set of templates, a technique pioneered by Daikon [10].

Equalities. For a variant expression e_1 and an (not necessarily variant) expression e_2[5] of conforming type, if $\overrightarrow{\mathcal{V}}_{e_1}^{q,c} = \overrightarrow{\mathcal{V}}_{e_2}^{q,c}$, AutoInfer introduces the candidate postcondition $e_1 = e_2$ for command c. *post_7* in Listing 1 is an example of inferred equality.

Linear relations. For any variant query expression e_1 of type *INTEGER*, AutoInfer tries to find a linear relation $e_1 = a \cdot e_2 + b$ or $e_1 = a \cdot \textbf{old}\ e_2 + b$, with $a, b \in \mathbb{Z}$, between e_1 and any other integer query expression e_2. To this end, AutoInfer uses linear regression techniques. *post_2* in Listing 1 is an example of inferred linear relation.

5.4 Validation

The validation step in the dynamic contract inference process of AutoInfer re-runs all tests in the test suite and checks the validity of all candidate contracts. A candidate is validated if it is not falsified by any execution. As in any dynamic inference technique, there is no absolute guarantee that a validated contract is indeed correct in absolute terms; however, a good test suite often provides a high confidence.

AutoInfer performs the validation step on all candidate contracts involving quantification (Section 5.1) and on basic contracts (Section 5.3). Implications inferred with decision trees (Section 5.2) require no validation, as a tree with confidence value of one matches by construction all data in the test suite. Before returning the final set of validated contracts to the user, AutoInfer performs a simple pruning which removes obviously redundant or trivial contracts.

[4] Typically, $m \ll n$.

[5] In this case, expressions can also be references to arguments of the command. The extension of the previous notation is straightforward, hence omitted for brevity.

6. EXPERIMENTAL EVALUATION

This section reports the results of experiments that applied AutoInfer to Eiffel implementations of several common data structures.

6.1 Experimental setup

Selection of classes. Table 1 lists the Eiffel classes implementing data structures used in the experiments; all the classes are from EiffelBase [9], the standard Eiffel data structure library. For each class, consider its size in lines of code (LOC), the number of commands (#CMD), queries (#QRY), precondition (#PRE) and postcondition (#POST) clauses provided by developers.

Setup and performance. All the experiments ran on a Windows 7 machine with a 2.53 GHz Intel dual-core CPU and 4 GB of memory. AutoInfer was the only computationally-intensive process running during the experiments.

The generation of the test suite (Section 3) ran for about 30 hours for each class; after the automated elimination of duplicated test cases, the final test suite consisted of 10 to 400 non-failing runs per command of each class. Following the test suite generation, the construction of the change profile (Section 4) took another 20–40 minutes for each command; finally, the actual contract inference phase ran for an average of 5.7 minutes per command.

6.2 Results and evaluation

The rightmost part of Table 1 summarizes the results of the contract inference experiments. The seventh column (#INF) reports the total number of inferred postcondition clauses of commands. We assess the quality of the inferred contracts according to their *soundness*, *completeness*, and *redundancy*. Future work (Section 8) will address the limitations evidenced by the experiments.

6.2.1 Soundness

Dynamic inference techniques cannot guarantee that every validated contract is sound. We manually inspected the contracts inferred to determine their soundness. The eighth column of Table 1 (%SND) reports the percentage of sound contract clauses. The percentage is quite high, and in line with the results obtained with other state-of-the-art dynamic inference tools for expressive specification inference (see Section 7).

Table 2 breaks down the total number of inferred clauses by type of assertion, revealing that clauses with implications (Section 5.2) have the highest percentage of unsoundness. This is the result of implications being the most sensitive to biased or incomplete test suites. As a representative example, consider command *merge_left* (*other*: *LINKED_LIST*) of class *LINKED_LIST*, which merges the list *other* to the left of the cursor in the current list. During the experiments, no pair of objects of class *LINKED_LIST* created by the test suite contained exactly the same elements, except for a few distinct instances of empty lists. AutoInfer noticed this accidental fact and correspondingly inferred the postcondition **old** *is_equal* (*other*) **implies** *is_empty* for *merge_left*. This postcondition is clearly unsound, as it is a mere reflection of an insufficiently varied test suite.

6.2.2 Completeness

Completeness addresses the question: is some contract clause missing? For each command c, we manually prepared a detailed postcondition π_c and demonstrated it complete according to a definition introduced in previous work [26], similar to the notion of relative completeness for algebraic specifications [15]. Intuitively, the postcondition of a command is complete if it describes the effects of the command, directly or indirectly, on every public query of the class.

We then used these complete postconditions as a yardstick to assess the completeness of inferred contracts: whenever the set of sound postcondition clauses inferred by AutoInfer for a command c entails the complete postcondition π_c, we consider the inferred contracts of c as complete as well. The method of [26] facilitates rigorous reasoning even without tool support, hence we have reasonable confidence in our assessment of completeness even if it was carried out manually. The ninth column (%CMP) of Table 1 lists the percentage of commands for which AutoInfer inferred a complete set of postconditions. Figures above 75% on average indicate that the inferred contracts are often very expressive and of high quality.

Let us also mention that the percentage of commands with a complete inferred contract drops to 67% if inference of implication expressions is disabled, and to 17% if inference of expressions with universal quantification is disabled. The few commands that admit a complete postcondition without quantification or implication are typically creation procedures (constructors).

Table 1 also shows that it is harder to generate complete postconditions for commands of classes implementing sets, arrays, and hash tables. These data structures have a more sophisticated semantics for commands than list data types; in a set, for example, the effect of adding an element changes according to whether the element is already present in the set or not. Correspondingly, AutoInfer sometimes does not generate a sufficiently varied set of objects or simply lacks a template needed to express a certain postcondition clause.

6.2.3 Redundancy

A postcondition clause is redundant if it can be inferred by the other clauses of the same command. More precisely, we introduce two notions of redundancy, of different strength. A postcondition clause is:

- A *repetition* if it is implied by the other postcondition clauses using only first-order reasoning and arithmetic; for example, **forall** i. $count - 1 + i =$ **old** $count + i$ and $count >$ **old** $count$ are both repetitions of $count =$ **old** $count + 1$.
- A *variation* if it is implied by the other postcondition clauses, *assuming knowledge of all complete contracts of queries* mentioned. For example, *post_7* in Listing 1 is a variation because it is implied by the combination of *post_2* with *post_3*: the *last* element corresponds to the element at position *count*, which equals **old** *count* + 1 after executing *extend*; this reasoning required knowledge of the exact contract of *last*.

Every repetition is a variation, while the converse does not hold in general. A variation which is not a repetition is likely to be a valuable form of redundancy, as it explicitly expresses the effect of a command on the value of queries not mentioned explicitly by other clauses, and makes the postcondition self-contained.

The eleventh and twelfth columns of Table 1 list the (largest) percentage of postcondition clauses that are variations

Table 1: Classes used in the experiments and inferred contracts statistics.

Class	LOC	#Cmd	#Qry	#Pre	#Post	#Inf	%Snd	%Cmp	%V	%R	%A
ARRAY	1463	15	32	69	107	235	94	66	80	0	74
HASH_TABLE	2021	14	36	49	102	200	94	50	74	8	22
LINKED_LIST	2000	26	34	70	91	378	98	84	76	2	74
LINKED_QUEUE	2099	6	20	78	94	61	96	100	62	2	84
LINKED_SET	2352	19	37	99	101	294	88	68	68	2	36
LINKED_STACK	2088	9	18	78	94	115	96	100	68	0	100
Total	*12023*	*89*	*177*	*373*	*589*	*1283*	*94*	*75*	*72*	*2*	*56*

of the others (%V) and the fraction of variations that are also repetitions (%R). The figures suggest that most of the redundant clauses are still useful and relevant: AutoInfer typically reports postconditions which are valuable, readable, and with little spurious or insignificant information.

As a further datum to assess informally the value of most inferred clauses, consider the last column (%A) of Table 1; it lists the percentage of programmer-written postcondition clauses that are a repetition of those automatically inferred. Given that an assertion written by programmers is certainly relevant, the abundance of inferred clauses subsuming programmer-written ones is evidence of mostly useful redundancies occurring.

Table 2: Type and soundness of inferred contracts.

Type	#Inf	#Snd (%)
Quantification	276	271 (98%)
Implication	242	220 (90%)
Equality	716	677 (94%)
Linear relation	49	49 (100%)

6.3 Limitations and threats to validity

The experiments demonstrate the effectiveness of the inference techniques of the paper. The techniques also have limitations, which future work will address (Section 8).

- We have so far applied the contract inference techniques to classes representing *data structure implementations*. These are often classes with carefully designed interfaces, including, in particular, a rich set of public queries. The techniques of the present paper may not apply as successfully to every class of programs; further experimentation is needed to clarify this aspect.
- The inference techniques focus on *postconditions of commands*. The empirical observation that programmers often write complete preconditions [25] justifies the focus on postconditions. The current techniques are not designed for inferring complex postconditions of queries (functions that do not change the object state). Improving this aspect will likely require new techniques, since most contract inference techniques treat queries as black boxes.
- All the experiments relied on *automatically generated test cases*: we do not know to what extent the effectiveness of the inference techniques is sensitive to how the test cases are generated.
- The experiments performed significantly better with two simplifications: consider only queries with at most one argument, and do not create nested data structures (e.g., lists of lists or sets of sets). Without these restrictions, AutoInfer must generate and analyze many more candidate contracts, resulting in a much longer running time for only a few more sound contracts. Optimizations and heuristics are needed to improve the performance in these scenarios.

7. RELATED WORK

Various approaches have targeted the problem of automatically inferring specifications of software components; a useful classification is between *static* and *dynamic* techniques.

7.1 Static techniques

Static techniques analyze the source code and infer specification elements by applying analytical techniques. The correct inference of all but the simplest classes of properties is undecidable, hence static techniques are usually sound but incomplete, and focus on tractable specifications.

For example, the abstract interpretation framework supports the inference of specifications such as interval constraints [3] or affine relations among variables [20]. Other static techniques can infer extended typing information [24], light-weight annotations ensuring the absence of out-of-bound and void pointer dereferencing errors [11], sequence diagrams describing the flow of information among objects [29], and invariants of loops [17, 12].

7.2 Dynamic techniques

Dynamic techniques summarize properties that are invariant over multiple runs of a program and suggest them as likely specification elements. The dynamic approach is very flexible in that it is applicable even if the source code is not available and does not require a sophisticated analytical framework. On the other hand, dynamic invariants come with no guarantee of soundness or completeness and their quality heavily depends on the choice of runs they summarize. Regardless of these limitations, dynamic techniques work quite well in practice.

To allow a closer comparison with the techniques of the present paper, the rest of this section focuses on applications of dynamic techniques to inferring specifications of components, in particular data structure implementations.

7.2.1 Finite-state behavioral specifications

Behavioral specifications summarize the behavior of a component as a finite-state automaton, constraining the sequences of commands that can occur.

Ammons et al. [1] build probabilistic finite-state automata which characterize order and data dependencies among invocations of public routines; their technique exploits machine

learning algorithms. Whaley et al. [33] complement dynamic techniques with static techniques to deduce common and illegal call sequences.

Dallmeier et al. [6] capture the relationship between commands and queries of a class with nondeterministic finite-state automata. Automata states abstract object states as a collection of queries; automata transitions are associated with commands. Xie et al. [34] build similar automata with multiple state abstractions, for example summarizing the information about the branch coverage. Lorenzoli et al. [21] extend the finite-state model with information about the values of attributes associated with each transition, and describe algorithms to learn such extended models.

Finite-state specifications are easy to read and analyze; on the other hand, they are usually coarse-grained and hence imprecise, especially to characterize the effects of a command. Algebraic specifications offer a more refined model.

7.2.2 Algebraic specifications

Algebraic specifications formalize component behavior in terms of functions on objects and equational axioms that describe the mutual relationship among such functions [15].

Henkel et al. [16] infer complex algebraic specifications of abstract data type implementations from automatically generated test cases. Ghezzi et al. [13] generalize finite-state models by means of graph-transformation systems, whose rules correspond to equations of algebraic specifications; their technique can build more accurate models than [16].

Algebraic specifications are quite expressive and appropriate to document abstract data type implementations. Their limitations include the difficulty of dealing with the *frame problem*; on the contrary, the present paper's techniques infer frame properties quite effectively. Another characteristic, shared by algebraic and finite-state models, is the fact that the specification refers to a whole component and does not describe the effects of each routine in isolation; this representation may hinder the integration of the specifications inferred in programming languages supporting user annotations.

7.2.3 Contract specifications

Ernst et al. [10] pioneered the usage of dynamic techniques with the Daikon tool. Daikon infers likely specifications in the form of annotations. It works with a pre-defined set of annotation templates which include relations among program variables and conjunctions thereof. The success of the approach has spawned derivative work extending the technique to target specific features or programs, such as polymorphic object-oriented languages [18, 4] or the consistency of data structure implementations [7], and to support more complex specifications.

A significant example of the latter are constraints in the form of implication. Daikon uses *splitting predicates* [10] as antecedents of implications; users provide such splitting predicates based on their understanding of the program. Follow-up work tried to automate the generation of splitting predicates, by extracting them from branch conditions in the program text [19] or by means of clustering [8]. Most of these proposals still lack a thorough evaluation; on the other hand, the present paper's solution (Section 5.2) proved to be quite effective at building expressive postconditions of data-structure routines.

Other work combined dynamic and static techniques, such as symbolic execution [5, 30], to boost the quality of the inferred specifications and to solve the problem of generating a set of test cases that guarantee an accurate inference.

The techniques of the present paper support several of these features with an original approach that leverages the availability of simple programmer-written contracts and is quite effective in terms of completeness and readability of the inferred specifications.

8. FUTURE WORK

Integration in environment. We feel the results are advanced enough to allow direct integration in the programming environment. We are building into the Eiffel Verification Environment (Eve) an advisory system that offers to programmers suggestions to improve reliability and facilitate verification. Inferred contracts are a prime example of such suggestions. It is important to point out here that we do not foresee adding assertions automatically; assertions and program correctness in general are a delicate matter, and the programmer must make the final decision. There may in fact be cases in which the inferred assertion will not match the programmer's intention, prompting the programmer to correct the program.

Soundness of inferred contracts. Some of the inferred contracts are not sound; this happens more often with assertions involving implications. We plan to identify unsound contracts by trying to automatically generate test cases that violate them. Soundness is clearly an essential requirement even though, in our expected scenario, the programmer will always make the final decision.

More forms for contracts. Inferring contracts for programs other than data structure implementations will require new forms for candidate assertions beyond universal quantifications and implications; for example, existential quantifications and expressions based on models that occur frequently in class abstractions, such as the bag [35].

Contracts of queries. Developing inference techniques for contracts of *queries*, as opposed to commands, is challenging. A query's postcondition describes the result returned in terms of elementary features of the class. For example, the postcondition of query *occurrences* (v: G) of *LINKED_LIST* must relate the nonnegative integer returned by the query to the elements equal to v stored in the list. We plan to rely on the notion of *model* of a class [26] to formalize the semantics of queries.

9. CONCLUSIONS

The present paper described advanced techniques for inferring complex contracts with clauses including universal quantification and implication. The techniques, which we have implemented in the AutoInfer tool, are based on dynamic analysis and data mining algorithms and are fully automatic. Experiments applying AutoInfer to standard Eiffel implementations of data structures such as linked list, queue, stack, array and hash table, inferred 75% of the complete postconditions with few unsound, redundant, or irrelevant clauses. These results suggest that AutoInfer can infer contracts of very high quality with little noise, hence providing accurate and readable documentation fully automatically.

The main consequence will be to provide programmers with relevant, non-trivial invariant suggestions as part of the routine feedback of the development environment, a key step

towards making the tool-supported quest for high reliability a standard part of the development process—"Verification as a matter of course".

AutoInfer is written in Eiffel and works on Eiffel classes. The techniques described in the paper are, however, applicable to other object-oriented languages that support contracts, such as JML for Java and Spec# for C#. The source code of AutoInfer, detailed experimental results and instructions to reproduce the experiments are available at:

`http://se.inf.ethz.ch/research/autoinfer`

Acknowledgments. The authors thank Yu Pei for suggestions to improve test case generation, Nadia Polikarpova for recommending the use of models, and Andreas Zeller for useful comments on a draft of the paper. Stefan Buchholz and Lucas S. Silva participated in early discussions and provided valuable feedback. This work has been partially funded by the Swiss National Science Foundation under the project SATS (SNF 200021-117995/1) and also benefited from funding by the Hasler foundation on related projects.

10. REFERENCES

[1] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL*, pages 4–16, 2002.

[2] P. Chalin. Are practitioners writing contracts? In *The RODIN Book*, volume 4157 of *LNCS*, page 100, 2006.

[3] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96, 1978.

[4] C. Csallner and Y. Smaragdakis. Dynamically discovering likely interface invariants. In *ICSE*, pages 861–864, 2006.

[5] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *ICSE*, pages 281–290, 2008.

[6] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. In *WODA*, pages 17–24, May 2006.

[7] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. Rinard. Inference and enforcement of data structure consistency specifications. In *ISSTA*, pages 233–243, 2006.

[8] N. Dodoo. Selecting predicates for conditional invariant detection using cluster analysis. Master's thesis, MIT EECS, 2002.

[9] http://freeelks.svn.sourceforge.net.

[10] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Eng.*, 27(2):99–123, 2001.

[11] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME*, volume 2021 of *LNCS*, pages 500–517. Springer, 2001.

[12] C. A. Furia and B. Meyer. Inferring loop invariants using postconditions. In *Fields of Logic and Computation*, volume 6300 of *LNCS*. Springer, 2010.

[13] C. Ghezzi, A. Mocci, and M. Monga. Synthesizing intensional behavior models by graph transformation. In *ICSE*, pages 430–440, 2009.

[14] N. Gupta and Z. V. Heidepriem. A new structural coverage criterion for dynamic detection of program invariants. In *ASE*, pages 49–59, 2003.

[15] J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Inf.*, 10:27–52, 1978.

[16] J. Henkel, C. Reichenbach, and A. Diwan. Discovering documentation for Java container classes. *IEEE Trans. Software Eng.*, 33(8):526–543, 2007.

[17] L. Kovács and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *FASE*, volume 5503 of *LNCS*, pages 470–485, 2009.

[18] N. Kuzmina and R. Gamboa. Extending dynamic constraint detection with polymorphic analysis. In *WODA*, pages 57–63, 2007.

[19] N. Kuzmina, J. Paul, R. Gamboa, and J. Caldwell. Extending dynamic constraint detection with disjunctive constraints. In *WODA*, pages 57–63, 2008.

[20] F. Logozzo. Automatic inference of class invariants. In *VMCAI*, volume 2937 of *LNCS*, pages 211–222. Springer, 2004.

[21] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *ICSE*, pages 501–510, 2008.

[22] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[23] B. Meyer, A. Fiva, I. Ciupa, A. Leitner, Y. Wei, and E. Stapf. Programs that test themselves. *IEEE Software*, pages 22–24, 2009.

[24] R. O'Callahan and D. Jackson. Lackwit: A program understanding tool based on type inference. In *ICSE*, pages 338–348, 1997.

[25] N. Polikarpova, I. Ciupa, and B. Meyer. A comparative study of programmer-written and automatically inferred contracts. In *ISSTA*, pages 93–104, 2009.

[26] N. Polikarpova, C. A. Furia, and B. Meyer. Specifying reusable components. In *VSTTE*, 2010.

[27] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[28] http://rapid-i.com/content/view/181/190.

[29] A. Rountev and B. H. Connell. Object naming analysis for reverse-engineered sequence diagrams. In *ICSE*, pages 254–263, 2005.

[30] N. Tillmann, F. Chen, and W. Schulte. Discovering likely method specifications. In *ICFEM*, volume 4260 of *LNCS*, pages 717–736. Springer, 2006.

[31] Y. Wei, S. Gebhardt, M. Oriol, and B. Meyer. Satisfying test preconditions through guided object selection. In *ICST*, pages 303–312, 2010.

[32] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *ISSTA*, pages 61–72, 2010.

[33] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *ISSTA*, pages 218–228, 2002.

[34] T. Xie, E. Martin, and H. Yuan. Automatic extraction of abstract-object-state machines from unit-test executions. In *ICSE*, pages 835–838, 2006.

[35] K. Yessenov, R. Piskac, and V. Kuncak. Collections, cardinalities, and relations. In *VMCAI*, volume 5944 of *LNCS*, pages 380–395. Springer, 2010.