Automatic Inference of Model Fields and their Representation [*]

Mihai Balint
Dept. of Computer and Software Engineering
Politehnica University of Timişoara, Romania
mihai@cs.upt.ro

Marius Minea
Dept. of Computer and Software Engineering
Politehnica University of Timişoara, Romania
marius@cs.upt.ro

ABSTRACT

Automatic mining or inference of formal specifications from program source code is a desirable goal for documentation and verification purposes. However, current approaches that generate invariants, pre- and post-conditions, procedure summaries and sometimes also class invariants have mostly focused on extracting specifications from concrete method bodies. Consequently, almost all results have a low level of abstraction that is very close to the analyzed source code.

We use JML model fields to raise the abstraction level of such automatically generated specifications, relying on the constraints imposed by behavioral subtyping. Starting from several derived classes we attempt to generate model fields for the supertype and `represents` clauses for each subtype. The relations between concrete and model fields are generated by checking the validity of predefined patterns against the specifications of subtype methods. Our prototype tool uses as inputs specifications generated with dynamic analysis (Daikon), identifies model fields and their representations, and generates specifications for supertype methods.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements / Specifications; D.2.4 [**Software Engineering**]: Software / Program Verification

[*]This work was partially supported by the Romanian National Research Council (CNCSIS) under project "Methods and Tools for Continuous Quality Assurance in Complex Software Systems" (PN II-IDEI 357/1.10.2007) and by the strategic grants POSDRU 6/1.5/S/13 (2008) of the Ministry of Labour, Family and Social Protection, Romania and POSDRU/21/1.5/G/13798, co-financed by the European Social Fund - Investing in People. We would also like to thank Google for funding a travel grant that enabled Mihai Balint to attend the conference and present this paper.

Keywords

automated specification mining, model field inference, specification inference, type inheritance, object-oriented software

1. INTRODUCTION

Automatic inference of formal specifications for programs is highly desired, as manual specification is intellectually intensive and error-prone. Software size and complexity not only makes specification inference more difficult, it can also lead to specifications that are themselves complex, difficult to understand and manage. Ideally, inferred specifications should raise the level of abstraction, expose commonalities in type hierarchies and observe representation independence.

For Java, the JML [19] specification language enables formal specification while preserving the inherent flexibility achieved in object-oriented software through encapsulation and implementation hiding. In JML specifications, information hiding is enabled through the use of model fields. These fields exist only in the specification and are used at verification time. Their role is to create an abstraction of a set of concrete fields, applicable to the given type and its subtypes. Using this abstraction, verifiers can reason about program behavior while implementation details remain hidden.

Thus, the most important use case for model fields together with type inheritance is to enable modularity and preserve encapsulation in the verification process. For example, Ruby and Leavens' subclassing method [24] uses only formally specified interfaces, without any information about the implementation (method bodies) of the base class.

While being a powerful specification instrument, model fields have been largely unused in the specification inference and program summarization communities. Hence, techniques that infer JML-like specifications, e.g., the pattern based approaches of Flanagan and Leino [14] and Ernst et al. [13], the later improvements by Kuzmina et al. [18], and the symbolic execution variations of Csallner et al. [10], Chandra et al. [7] have been most successful when the contents of the method body is available.

Conversely, when method bodies are unavailable (e.g., for abstract classes and interfaces), specifying abstract members for class hierarchies has been mostly handled by quantification of subclass-specific information, e.g., the attempt of Goldstein et al. [15] at specification refactoring, which sometimes results in methods with unsatisfiable preconditions.

When only interface definitions are available, specification mining research has found that generating automata to model behavior is more useful for the goals of software engineering, hence the proliferation of techniques like the ones presented by Alur et al. [1], Shoham et al. [25], Weimer et al. [26] and Dallmeier et al. [11]. Previously, we have also proposed an automata based approach [2] which inferred super-class specifications; here, we use these ideas to improve the expressiveness of inferred JML specifications in the case of abstract classes and interfaces.

In this paper, we explore the idea of model field inference. Starting from several classes with method specifications, we automatically infer model fields and formulas for the `represents` clauses of each derived class. Model fields allow us to rewrite method preconditions and summaries at a more abstract level, with fewer references (ideally absolutely no references) to concrete fields. Next, we obtain method specifications for the base class that observe the rules of behavioral subtyping. The end result is a set of specifications which are simpler and easier to understand, and where model fields preserve encapsulation and expose the common behavior of the class hierarchy.

2. MODEL FIELD INFERENCE

In this section we briefly describe model fields, then explain the procedure used to validate potential `represents` relations against subtype specifications.

2.1 JML Model Fields

Model fields (also called abstract variables in [20]) enable the specification of behavior without referencing actual implementation-specific data. This feature preserves encapsulation and allows formulation of specifications at a higher level of abstraction.

Each model field has a representation which defines its relation to real fields or other model fields. In JML this is specified using `represents` clauses which can have two forms, a functional form that directly defines the model field using an equality =, or a relational form indicated by the keyword `\such_that`. The functional form defines the representation of the model field as a function of other (real or model) fields and is the most commonly used. The relational form is more general and defines the possible values of a model field given a certain program state using the Boolean expression specified after the `\such_that` keyword. A complete formal semantics for model fields, based on Hilbert's ϵ-terms is under development [4].

An example of model field declaration and representation within a hierarchy is presented in Listing 1 using the JUnit example from [3] annotated with JML specifications.

2.2 Inferring model field representations

Overall, the inference starts by targeting a certain base type (either base class or interface) and assuming the existence of a model field with representations in each of the derived types. The `represents` relation is chosen from a list of predefined patterns and is validated against the specifications of all methods defined by the supertype and specified in terms of concrete type fields (as typically produced with current automated specification mining techniques).

Formally, given a base type T and subtypes S_i, each declaring a tuple of real fields x_i and a relation $R_i(mf, x_i)$ which defines a potential representation of model field mf in each subtype, we can check the validity of each R_i by verifying the satisfiability of formula (1) against the preconditions P_{S_i} and the procedure summaries Q_{S_i} of the subtype implementations m_i for all methods m declared by T.

$$\begin{aligned}&(\bigwedge_{S_i \leq T} P_{S_i}(x_i, y_i, p) \wedge R_i(mf, x_i)) \;\Rightarrow \\ &\bigwedge_{S_i \leq T} (\forall x_i'.Q_{S_i}(x_i, y_i, p, x_i', y_i', p') \Rightarrow \exists mf'.R_i(mf', x_i'))\end{aligned} \quad (1)$$

We assume that the procedure summaries include the precondition constraints. We distinguish (tuples of) concrete fields x_i that appear in the model field relation R_i and the remaining concrete fields y_i that don't. Preconditions and summaries are expressed in terms of fields x_i, y_i and parameters p. Primed variables correspond to the post-state. Formula (1) validates a potential correlation between the real fields of all implementations, ensuring that for any outcome of real fields x_i' there is a corresponding value mf' of the model field which preserves the desired relation R_i.

By assuming the conjunction of preconditions we ensure the weakest possible safe context from which all method implementations can be called (behavioral subtyping [21]). Also, we avoid unsafe strengthening of the postcondition by checking the validity of the model field representation in the post-state independently for each subtype. Subtyping constraints are used here because our goal is to raise subtype specs to an abstraction level which enables their use as supertype specs, thus we only care about the validity of `represents` relation within these requirements.

We obtain the actual `represents` relations using a repository of possible patterns which we instantiate with different real fields declared by subtypes. The complete algorithm is listed in pseudo-code below.

Algorithm 1 Find model field representations

```
for all $P \in$ pattern repository do
  for all $S_i$ subtypes of $T$ do
    for tuples $F$ of $S_i$ fields matching $P$ do
      $R_{i,j} \leftarrow$ instantiate $P$ with $F$
      add $R_{i,j}$ as a potential represents relation for $S_i$
    end for
  end for
end for
for all tuples $M$ of relations $R_{i,j}$ across the set $\{S_i\}$ do
  keep $M$ if all its relations satisfy equation (1)
end for
```

At the end of Algorithm 1 we obtain all `represents` relations which have been validated for the given set of subtypes. In some cases, instantiating a relation pattern and validating it may be partly merged, as is the case with linear arithmetic relations for which after instantiation we must infer valid numerical coefficients.

Depending on the variety of patterns in the repository, Algorithm 1 may generate model fields and `represents`relations which model the same program state at different levels of abstraction, according to which patterns *fit*. This may result in *specification spam*, with multiple potential specifications for a super-class. For safety reasons, we do not throw away any of the generated fields and propose that (1) either the program maintainer makes an informed choice for a certain version or (2) when a program property must be proved, all specifications should be tried and the ones which satisfy the property be attached as preconditions to the proof.

Because inheritance is used in Java both for subtyping and implementation reuse, we require that each method which was given an implementation within the supertype be specified separately for each subtype. This requirement is due to the fact that such methods may polymorphically alter the specific state of subtypes. Omitting these modifications from specifications hinders the effectiveness of our approach, however this type of specification only makes sense in the context of subtypes as it refers to subtype-specific variables.

3. ABSTRACTION

In this section, we discuss the methods by which the generated `represents` relations for model fields can be used as vehicles for abstraction. This includes the inference of more abstract specifications for the subtypes, eliminating concrete fields, and the generation of formal specifications for the supertype, using the behavioral subtyping relation [21].

3.1 Abstraction of subtype specifications

Once a model field and valid `represents` relations are identified we can remove implementation-specific details from subtype specifications. The goal is to obtain method specifications (preconditions and procedure summaries) that reference as few fields declared in the subtype as feasible, employing instead model fields where possible.

Formally, for preconditions this means computing the quantified formula below, where $R(mf, x)$ is the `represents` relation for model field mf, $P_S(x, y, p)$ is the precondition expressed in terms of subtype fields and method arguments, and $P'_S(mf, y, p)$ is the precondition expressed in terms of model fields and method arguments.

$$P'_S(mf, y, p) = \exists x.R(mf, x) \wedge P_S(x, y, p)$$

For the procedure summary $Q'_S(mf, p, mf', p')$, the formula is similar, while accounting for both pre-state and post-state versions of the fields and the method parameters.

$$Q'_S(mf, y, p, mf', y', p') = \exists x, x'.R(mf, x) \wedge R(mf', x') \wedge Q_S(x, y, p, x', y', p')$$

In both of the above formulas, existential quantification is only performed over the concrete fields x that appear in the `represents` relation R; the results may still depend on other (possibly subtype specific) fields y.

We can perform a further round of abstraction by replacing the equivalence relations with implications. To maintain safety, we strengthen the precondition (equation 2) while weakening the procedure summary (equation 3).

$$P'_S(mf, y, p) \Rightarrow \exists x.R(mf, x) \wedge P_S(x, y, p) \tag{2}$$

$$Q'_S(mf, y, p, mf', y', p') \Leftarrow \exists x, x'.R(mf, x) \wedge R(mf', x') \wedge Q_S(x, y, p, x', y', p') \tag{3}$$

In practice, abstractions could involve eliminating concrete fields y which R does not relate to the model field, using universal quantification to strengthen the precondition and existential quantification to weaken the method summary.

3.2 Generating supertype specifications

For each method defined in the supertype, we use the preconditions and procedure summaries of its implementations in the various subtypes to generate an appropriate specification in the subtype, while observing the rules of behavioral subtyping. For a method m defined in supertype T and implemented in two subtypes S_1 and S_2, behavioral subtyping enforces the following constraints on the preconditions and postconditions of the implementations of m:

$$P_T \Rightarrow (P_{S_1} \wedge P_{S_2})$$
$$(Q_{S_1} \vee Q_{S_2}) \Rightarrow Q_T$$

To generate specifications for m as defined in the supertype, we replace the implications with equivalence. In general, for N subtypes we define the precondition and summary of method m using equations 4 and 5 respectively.

$$P_T(mf, p) = \bigwedge_{S_i \leq T} \forall y_i.P_{S_i}(mf, y_i, p) \tag{4}$$

$$Q_T(mf, p, mf', p') = \bigvee_{S_i \leq T} \exists y_i, y'_i.Q_{S_i}(mf, y_i, p, mf', y'_i, p') \tag{5}$$

After the application of the generalization technique from section 3.1 it is usually the case that subtype specific fields (i.e., y_i) are still referenced within the specification. This is caused by implementation dissimilarities which could not be modeled in a uniform manner using the selected `represents` relations. These fields have no meaning in the context of the supertype and are eliminated using quantification.

At this stage, the precondition P_T of method m for type T may become unsatisfiable either because of the universal quantification or because there exists no value of mf which satisfies all subtype preconditions. This is a sign that the method cannot be uniformly invoked across all N subtypes and as such this hierarchy violates the Liskov Substitution Principle [21]. Empirical observations [22] have shown this situation to be quite common but also offer an insight: the offending subtypes are either few or are clustered in places where the hierarchy lacks intermediate types. In this latter case the goal consists of finding subsets of subtypes for which P_T is satisfiable and adding new intermediate types to the hierarchy. Similarly, Q_T may become universally satisfiable, a case which also points toward poor hierarchy structuring and potential software design problems.

Sometimes Java inheritance is used simply for code reuse by placing common code within a base class. It can be difficult to differentiate between inheritance for code reuse and inheritance for subtyping. In the former case the results of our approach would be meaningless, as such "derived" classes are not intended for use through their base class interface. A good heuristic for identifying code reuse through inheritance relies on the observation that the base class is not used for any other purposes (i.e., for variable declarations). Unfortunately, in practice these two uses of Java inheritance are sometimes mixed: some base class methods exist purely for code-reuse, while others belong to the type hierarchy. These cases occur mainly because design discipline is not enforced.

4. PROTOTYPE IMPLEMENTATION

We have implemented our approach in kaisō, a prototype tool which given a Java class attempts to identify in non-abstract derived classes the representations of a model field. If valid representations are found, the specifications of all methods declared by the base class are given in terms of the identified model field and the visible real fields (i.e., fields not declared within derived classes).

The tool uses as input the procedure summaries generated through dynamic analysis with Daikon [13] with a modified front-end for bytecode instrumentation and execution monitoring. We are currently developing an interface for procedure summaries generated through symbolic execution and other static analyses.

Our prototype supports as relation patterns equality and linear arithmetic relations with arbitrary numeric coefficients. We have developed a module which identifies the coefficients of a potential linear relation by generating tests with concrete values satisfying the given procedure summaries.

For linear arithmetic relations, validity checking and coefficient discovery are fused since test generation relies on many of the same assertions as validation. Coefficient identification relies on proving that real fields from derived types change in linearly related ways. Solving systems of linear equations, we identify the real field coefficients, while the constant field is found using the specifications of other methods (usually field setters). The advantage of this technique is that its performance is predictable: the number of required tests depends on the number of analyzed real fields. The disadvantages are (1) it requires methods that mutate object state and (2) the constant coefficient can only be identified using a different method. These limitations may be overcome by using copy-on-write methods instead of mutator methods, if we identify the source and destination objects.

To achieve a high level of automation and performance our current implementation uses the Yices SMT solver [12] for test generation as well as the rest of our proving requirements. The drawback is that the expressiveness of the used specifications and `represents` relations is limited. To successfully generate specifications for supertypes we currently limit `represents` relation patterns to invertible functions (or invertible systems of equations) which can be trivially used to define real fields in terms of model fields using functions that can then be reused to obtain equations 2 and 3. This limitation could be avoided using a more general theorem prover to replace the SMT solver.

4.1 Experiments

To test our prototype implementation we have used the test suite developed as a JUnit showcase in [3] to generate specifications for two classes and model fields for the interface implemented by the two classes.

Listing 1 depicts part of the annotated `IMoney` type hierarchy. The `IMoney` interface specifies the model field `coins` which is an abstraction of the total number of money units contained by a certain object. One implementation, the `Money` class, holds a certain amount of money in a given currency and provides a functional representation for the `coins` model field. The second implementation, `MoneyBag`, holds a list of `IMoney` objects and provides a relational representation for `coins`. In this example the relation is total and one-to-one and could trivially be written in functional form; however, this need not be the case in general.

The test suite was instrumented and executed using the Daikon invariant detector. The generated specifications for `Money` and `MoneyBag` were then analyzed using kaisō which correctly identified a model field that could be used in the `IMoney` interface and its `represents` relations in `Money` and `MoneyBag`. The results are depicted in Table 1.

Using the inferred `represents` relations and assuming that the inferred model field is named `coins` it is trivial to refine

```
public interface IMoney {
  //@ public instance model int coins;

  /* Adds a money to this money. */
  /*@ ensures \result.coins =
    @           this.coins + m.coins;
    @*/
  public IMoney add(IMoney m);
...

public class Money implements IMoney {
  private int fAmount;
  private String fCurrency;
  //@ represents coins = fAmount;
...

public class MoneyBag implements IMoney {
  private List<Money> fMonies;
  /*@ represents coins  \such_that
    @   coins == (\sum int i;
    @              0 <=i && i<fMonies.size();
    @              fMonies.get(i).fAmount
    @   );
    @*/
...
```

Listing 1: Money hierarchy example

Table 1: ***Model field relation for the*** `IMoney` ***hierarchy***

supertype: `IMoney`	`represents` relation for *mf*
subtype: `Money`	$mf - fAmount = 0$
subtype: `MoneyBag`	$mf - sum(fMonies.get(i).fAmount) = 0$

the initially generated specifications by replacing references to `fAmount` and `sum(fMonies.get(i).fAmount)` with references to `coins`. Sample resulting specifications for `MoneyBag` are shown in table 2 compared with the results obtained with the *Turnip* tool of Kuzmina and Gamboa [17].

In the specifications obtained using *Turnip* for method `add(IMoney m)`, references to subtype fields are guarded using type checks, although conceptually the specifications express the same thing. Using the `coins` model field within these specifications we avoid referencing real fields from `Money` and `MoneyBag`, which means that the type checks become redundant and can be removed, reducing the postcondition of `add(IMoney m)` to a single line.

For the `subtract(IMoney m)` method, Daikon filters out some potential invariants because the number of tests is insufficient for its default confidence threshold. This causes the specification generated by *Turnip* to cover only three of the four possible combinations of parameter and return types. By considering all invariants which have not been invalidated, kaisō is able to produce an improved postcondition which covers all four type cases.

4.2 Validity

As with all automated techniques, the meaning of the generated model fields may be questionable. The `represents` relations from each derived class only provide limited hints, however, the conceptual abstraction step remains a burden for the developer. In the `Money` and `MoneyBag` example, does it make sense to sum up amounts of different currencies? And what does that sum represent? In our opinion such questions are better left for software architects and designers

Table 2: *Specifications for the* `MoneyBag` *class*

Method	`IMoney add(IMoney m)`
Turnip Postcondition	$(m.class = MoneyBag \wedge return.class = MoneyBag) \Rightarrow$ $(m.fMonies[].sum - return.fMonies[].sum + this.fMonies[].sum = 0)$
(from [17])	$(m.class = MoneyBag \wedge return.class = Money) \Rightarrow$ $(m.fMonies[].sum - return.fAmount + this.fMonies[].sum = 0)$
	$(m.class = Money \wedge return.class = MoneyBag) \Rightarrow$ $(m.fAmount - return.fMonies[].sum + this.fMonies[].sum = 0)$
	$(m.class = Money \wedge return.class = Money) \Rightarrow$ $(m.fAmount - return.fAmount + this.fMonies[].sum = 0)$
kaisō Postcondition	$m.coins \ - \backslash result.coins \ + this.coins \ = 0$
Method	`IMoney multiply(int factor)`
Turnip Postcondition	$(return.class = MoneyBag) \Rightarrow$ $(return.fMonies[].sum = this.fMonies[].sum * factor)$
	$(return.class = Money) \Rightarrow$ $(return.fAmount = this.fMonies[].sum * factor)$
kaisō Postcondition	$\backslash result.coins \ = this.coins \ * factor)$
Method	`IMoney subtract(IMoney m)`
Turnip Postcondition	$(m.class = MoneyBag \wedge return.class = MoneyBag) \Rightarrow$ $(m.fMonies[].sum + return.fMonies[].sum - this.fMonies[].sum = 0)$
	$(m.class = MoneyBag \wedge return.class = Money) \Rightarrow$ $(m.fMonies[].sum + return.fAmount - this.fMonies[].sum = 0)$
	$(m.class = Money \wedge return.class = Money) \Rightarrow$ $(m.fAmount + return.fAmount - this.fMonies[].sum = 0)$
kaisō Postcondition	$m.coins + \backslash result.coins - this.coins = 0$

to answer. While an automated approach cannot provide an answer to these questions, we do obtain valid specifications which are more precise and compact than previous work.

Our approach relies on specifications provided from external sources. While we stress the fact that our approach complements existing automatic techniques for specification extraction, we do not strictly depend on them, as we can equally well accommodate developer specifications. However, in software development processes which use formal methods, specifications are usually written prior to code. We envision that our approach would be most useful in the maintenance and reengineering phase of the software process. The dependency on existing specifications also means that all their limitations (incompleteness, unsoundness) would also affect the reliability of our results. We make no attempt to identify or resolve any of these problems and assume that they have been accounted for before our tools are employed.

5. RELATED WORK

The specification problem we address requires the availability of method specifications (possibly inferred automatically); in turn we produce refined specifications which can be used by approaches to verification relying on model fields.

Specification mining.

Existing approaches typically rely either on static program analysis or on dynamic execution analysis.

Large scale static summarization for Java programs was recently employed for bug finding [7]. The results provide little information regarding the compactness, readability or modularity of the generated summaries. However, relying on static analysis of program source code means that their summaries have a low level of abstraction and might benefit from our refining approach.

Employing a different tactic, [25] and [16] analyze API and generate transitions systems which model common usages. While also generating specifications for abstract classes and interfaces, their results are best suited for verifying temporal sequencing properties of programs.

Dynamic analysis methods often use patterns of invariants which they validate against real program executions. Our approach also relies on constraint patterns, but employs them to refine existing specification by introducing more abstract model fields. Our prototype uses the Daikon invariant detector [13] as an initial provider of specifications which we then analyze and improve, complementing Daikon's limitations regarding interfaces and abstract methods. An initial attempt to overcome these limitations was presented in [17]. It relies on polymorphic analysis and generates specifications which are guarded with type checks. As we have shown, our approach enhances these results by rendering type checks redundant and making specifications more compact.

Verification using model fields.

Model fields are the central vehicle employed for specification abstraction by our method.

In the verification community, model fields have raised some challenges mainly because their initial definition was informal. Early attempts like the one presented by Müller [23] dealt only with the functional representation of model fields. Others handled only a restricted form of the relational representation [6, 9, 8]. Only recently have verification tools started to employ complete model field verification [5] and a complete language independent semantics has been proposed [4]. Our approach can enhance the usability of these verification techniques by providing them with specifications of increased relevance.

6. CONCLUSIONS

We have presented an approach to specification inference that focuses on behavioral subtyping and employs model fields to raise the abstraction level of method specifications. Given a set of subtypes, we use a library of relational patterns to automatically infer model fields and their representations in each type, while observing behavioral subtyping constraints. Subsequently, we can generate specifications for the common supertype which use model fields as an abstract representation of the concrete fields in the subtypes. Our specifications are thus more abstract and potentially more understandable than those produced by typical specification inference approaches alone.

This approach preserves encapsulation as a desirable feature of object-oriented design. Moreover, the success in generating usable model fields and supertype specifications is an indicator of a discipline that uses inheritance for subtyping rather than merely for code reuse and thus correlates well with methods that evaluate design quality.

7. REFERENCES

[1] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *32nd Symp. on Principles of Programming Languages*, POPL'05, pages 98–109, 2005.

[2] M. Balint. Automatic inference of abstract type behavior. In *Int. Conf. on Automated Software Engineering*, ASE '10, pages 499–504, 2010.

[3] K. Beck and E. Gamma. *Test-infected: programmers love writing tests*, pages 357–376. Cambridge University Press, 2000.

[4] B. Beckert and D. Bruns. Formal semantics of model fields in annotation-based specifications inspired by a generalization of Hilbert's ϵ terms. To appear, 2011.

[5] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.

[6] C.-B. Breunesse and E. Poll. Verifying JML specifications with model fields. In *Proceedings of the Fifth ECOOP Workshop on Formal Techniques for Java-like Programs*, 2003.

[7] S. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'09, pages 363–374, 2009.

[8] Y. Cheon, G. Leavens, M. Sitaraman, and S. Edwards. Model variables: cleanly supporting abstraction in design by contract. *Software: Practice and Experience*, 35:583–599, May 2005.

[9] D. R. Cok. Reasoning with specifications containing method calls and model fields. *Journal of Object Technology*, 4(8):77–103, 2005.

[10] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: dynamic symbolic execution for invariant inference. In *International Conference on Software engineering*, ICSE'08, pages 281–290, 2008.

[11] V. Dallmeier, A. Zeller, and B. Meyer. Generating fixes from object behavior anomalies. In *Int. Conf. on Automated Software Engineering*, 2009.

[12] B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In *Proceedings of the* 18^{th} *International Conference on Computer Aided Verification*, volume 4144 of *LNCS*, pages 81–94. Springer, 2006.

[13] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programm.*, 69(1-3):35–45, 2007.

[14] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, FME'01, pages 500–517, 2001.

[15] M. Goldstein, Y. A. Feldman, and S. Tyszberowicz. Refactoring with contracts. In *Proceedings of the Agile Conference*, pages 53–64. IEEE Computer Soc., 2006.

[16] N. Gruska, A. Wasylkowski, and A. Zeller. Learning from 6,000 projects: lightweight cross-project anomaly detection. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA'10, pages 119–130, 2010.

[17] N. Kuzmina and R. Gamboa. Extending dynamic constraint detection with polymorphic analysis. In *the 5th International Workshop on Dynamic Analysis*, WODA'07. IEEE Computer Society, 2007.

[18] N. Kuzmina, J. Paul, R. Gamboa, and J. Caldwell. Extending dynamic constraint detection with disjunctive constraints. In *the* 6^{th} *International Workshop on Dynamic Analysis*, WODA'08, pages 57–63, 2008.

[19] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, D. M. Zimmerman, and W. Dietl. *JML Reference Manual, Draft Revision 2344*, Feb. 2011. Available at http://www.jmlspecs.org/documentation.shtml.

[20] K. R. M. Leino. Data groups: specifying the modification of extended state. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'98, pages 144–153, 1998.

[21] B. H. Liskov and J. M. Wing. Behavioural subtyping using invariants and constraints. In *Formal methods for distributed processing: a survey of object-oriented approaches*, pages 254–280. Cambridge University Press, 2001.

[22] P. F. Mihancea and R. Marinescu. Discovering comprehension pitfalls in class hierarchies. In *European Conference on Software Maintenance and Reengineering*, pages 7–16, 2009.

[23] P. Müller. *Modular specification and verification of object-oriented programs.* Springer-Verlag, 2002.

[24] C. Ruby and G. T. Leavens. Safely creating correct subclasses without seeing superclass code. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'00, pages 208–228, 2000.

[25] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. In *International Symposium on Software Testing and Analysis*, ISSTA'07, pages 174–184, 2007.

[26] W. Weimer and N. Mishra. Privately finding specifications. *IEEE Transactions on Software Engineering*, 34(1):21–32, 2008.