Second-Order Constraints in Dynamic Invariant Inference

Kaituo Li
Computer Science Dept.
University of Massachusetts
Amherst, MA 01003, USA

Christoph Reichenbach
Institut für Informatik
Goethe University Frankfurt
Frankfurt am Main 60054,
Germany

Yannis Smaragdakis
Dept. of Informatics
University of Athens
Athens 15784, Greece

Michal Young
Computer Science Dept.
University of Oregon
Eugene, OR 97403, USA

ABSTRACT

The current generation of dynamic invariant detectors often produce invariants that are inconsistent with program semantics or programmer knowledge. We improve the consistency of dynamically discovered invariants by taking into account higher-level constraints. These constraints encode knowledge *about* invariants, even when the invariants themselves are unknown. For instance, even though the invariants describing the behavior of two functions *f1* and *f2* may be unknown, we may know that any valid input for *f1* is also valid for *f2*, i.e., the precondition of *f1* implies that of *f2*. We explore techniques for expressing and employing such consistency constraints to improve the quality of produced invariants. We further introduce techniques for dynamically discovering potential second-order constraints that the programmer can subsequently approve or reject.

Our implementation builds on the Daikon tool, with a vocabulary of constraints that the programmer can use to enhance and constrain Daikon's inference. We show that dynamic inference of second-order constraints together with minimal human effort can significantly influence the produced (first-order) invariants even in systems of substantial size, such as the Apache Commons Collections and the AspectJ compiler. We also find that 99% of the dynamically inferred second-order constraints we sampled are true.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Class invariants*; D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*

General Terms

Reliability,Verification

Keywords

Daikon, dynamic invariant inference, second-order constraints

ESEC/FSE '13, August 18-26, 2013, Saint Petersburg, Russia
Copyright 2013 ACM 978-1-4503-2237-9/13/08 ...$15.00.

1. INTRODUCTION AND MOTIVATION

Systematically understanding program properties is a task at the core of many software engineering activities. Software testing, software documentation, and software maintenance benefit directly from any significant advance in automatically inferring program behavior. *Dynamic invariant detection* is an intriguing behavior inference approach that has received considerable attention in the recent research literature. Tools such as Daikon [6], DIDUCE [14], and DySy [5] monitor a large number of program executions and heuristically infer abstract logical properties of the program, expressed as invariants.

Dynamic invariant inference is fundamentally a search in the space of propositions over program variables for the tiny subset of propositions that are both true (on all executions) and relevant (to the programmer, or to some client analysis such as a theorem prover or test data generator). Any such search requires a way of generating candidate propositions and a way of evaluating them. Even limiting consideration to propositions of fixed length, over a particular vocabulary of operations and relations among variables, the space is enormous, and for a long time it seemed folly to even attempt such a search based on dynamic program execution. Daikon [6] was the first demonstration that a generate-and-test tactic could be feasible, generating a moderately large set of candidate propositions from syntactic templates and quickly pruning the vast majority as they were refuted by program execution.

In this work we enhance the dynamic invariant detection approach by including second-order constraints. The invariants inferred should be consistent with these constraints. More specifically, we identify a vocabulary of constraints on inferred invariants. We call these constraints "second-order" because they are constraints over constraints: they relate classes of invariants (first-order constraints). Such second-order constraints can be known even though the invariants are unknown. (To avoid confusion, we try to consistently use the term "constraints" for second-order constraints, as opposed to the first-order "invariants", although the term "second-order invariants" would have been equally applicable.) Our vocabulary includes concepts such as *Subdomain* (one method's precondition implies that of another); *Subrange* (one method's postcondition implies that of another); *CanFollow* (one method's postcondition implies the precondition of another); *Concord* (two methods satisfy a common property for the common part of their domains, although some inputs for one may be invalid for the other); and more.

Such constraints arise fairly naturally. Two methods may be applicable at exactly the same state of an input. For instance, the `top` and `pop` methods of a stack data structure will have constraints:

```
Subdomain(top, pop)
Subdomain(pop, top)
```

This signifies that their preconditions are equivalent. As another example, a method may be a specialized implementation of another (e.g., a matrix multiplication with a faster algorithm, applicable to upper triangular matrices only, relative to a general matrix multiplier). This induces a `Subdomain(triangMatMul(), matMul())` constraint. Similarly, a method may be preparing an object's state for other operations, as in the common case of `open` and `read`: `CanFollow(open(), read())`.

The purpose served by second-order constraints is dual:

- First, when the constraints are discovered dynamically, without any human effort expended, they can serve as a concise and deeper documentation of program behavior—e.g., a single second-order constraint can distill the meaning of many Daikon invariants. This can be seen as an effort to generalize the observations that the Daikon system already makes via sophisticated post-processing. For instance, state-machine behavior (e.g., calls to method `foo` can follow calls to method `bar`) can be documented in the vocabulary of second-order constraints.
- Second, when the (second-order) constraints are either supplied by the programmer or vetted by the programmer (among candidate constraints suggested automatically) they can help enhance dynamically inferred (first-order) invariants by effectively cross-checking invariants against others. In this way, the noisy inferences caused by having a limited number of dynamic observations are reduced: the second-order constraint links the invariant at a certain program point with a larger set of observed values from different program points. In exchange for modest specification effort, second-order constraints can eliminate inconsistencies and produce more relevant and more likely-correct invariants.

In more detail, the main contributions of our paper are as follows:

- We identify and describe the idea of second-order constraints that help improve the consistency and relevance of dynamically inferred invariants.
- We define a vocabulary of common second-order constraints.
- We present a generalization of Daikon's dynamic inference approach to second-order constraints, by appropriately defining the Daikon "confidence" metric for second-order properties.
- We discuss how second-order constraints can be implemented, how their enforcement is affected by properties of the dynamic invariant inference system (we identify *monotonicity* as a particularly important property). We provide an implementation of the constraints in our vocabulary in the context of the Daikon system.
- We provide experimental evidence for the value of second-order constraints and their dynamic inference. We evaluate our approach both in controlled micro-benchmarks and in realistic evaluations with the Apache Commons Collections and the AspectJ compiler. A low-cost effort to write constraints (e.g., a few hours spent to produce 26 constraints) results in significant differences in the dynamically inferred invariants. Furthermore, the vast majority (99%) of a random sample among our dynamically inferred second-order constraints are found to be true.

2. INVARIANTS AND CONSTRAINTS

We begin with a brief background on dynamic invariant inference, and subsequently present our vocabulary of second-order constraints. For illustration purposes, our discussion is in the context of Java and the Daikon invariant inference tool. This is the most mainstream combination of programming language and dynamic invariant inference tool, but similar observations apply to different contexts and tools.

2.1 Daikon and Dynamic Invariant Inference

Daikon [6,24] tracks a program's variables during execution and generalizes their observed behavior to invariants. Daikon instruments the program, executes it (for example, on an existing test suite or during production use), and analyzes the produced execution traces. At each method entry and exit, Daikon instantiates some three dozen invariant templates, including unary, binary, and ternary relations over scalars, and relations over arrays (relations include linear equations, orderings, implication, and disjunction). For each invariant template, Daikon tries several combinations of method parameters, method results, and object state. For example, it might propose that some method `m` never returns null. It later discards those invariants that are refuted by an execution trace—for example, it might process a situation where `m` returned null and it would therefore discard the above invariant. Daikon summarizes the behavior observed in the execution traces as invariants and generalizes it by proposing that the invariants hold in all other executions as well. Daikon can annotate the program's source code with the inferred invariants as preconditions, postconditions, and class invariants in the JML [18] specification language for Java.

2.2 A Vocabulary of Constraints

To express rich constraints on invariants, we use a vocabulary including constraints such as *Subdomain* and *Subrange*. These constraints are specified explicitly by the user of the invariant inference system by employing an annotation language. Furthermore, we give the user an opportunity to express one more useful kind of information regarding inferred constraints: a set of "care-about" program variables and fields that are allowed in inferred invariants. Our vocabulary includes the following concepts:

- *OnlyCareAboutVariable(⟨var⟩)* and *OnlyCareAboutField(⟨fld⟩)*: These annotations instruct the invariant inference system that the derived invariants of a method should only contain the listed method formal argument variables or class fields. This prevents the system from deriving facts irrelevant to the property under examination, which would also be more likely to be erroneous. Additionally, it allows restricting invariants so that complex constraints can be expressed. The *OnlyCareAbout...* constraints are not consistency constraints on invariants, but they are essential background for use together with the consistency constraints that follow.
- *Subdomain(m1,m2)*: The precondition of method *m1* implies the precondition of method *m2*. For instance, the author of a matrix library may specify the constraint:

  ```
  Subdomain(processDiagonal, processUpperTriangular)
  ```

  This means that method `processDiagonal`, which works on diagonal matrices, has a precondition that implies that of method `processUpperTriangular`, which works on upper triangular matrices: the latter method is applicable wherever the former is. (Matrix libraries often have a large number of related operations, in terms of interface and applicability, but with different implementations for performance reasons [13]. In this section, we draw several examples from this domain for illustration purposes.)

  Preconditions are not only expressible on method parameters but also on member fields of an object. E.g., one may specify that the preconditions of method `append` imply those of method `write`: If the object is in an appendable state, it can also be written to.

In order to specify useful *Subdomain* constraints, one may need to limit the inferred invariants by employing the *OnlyCareAboutVariable* and *OnlyCareAboutField* annotations presented earlier. (This also holds for many of the later constraints.) For instance, methods *m1* and *m2* may be on separate objects. Thus, their invariants would normally include distinct sets of variables. Yet if we specify their common variables with *OnlyCareAboutVariable*, then the *Subdomain* constraint can be applicable.[1]

- *Subrange(m1,m2)*: This constraint is analogous to *Subdomain* but for postconditions. It means that the postcondition of method *m1* is stronger than (i.e., implies) that of *m2*. Again, the condition may apply to parameter or return values—e.g., a method with postcondition `isDiagonal(return)` is related with *Subrange* to one with postcondition `isUpperTriangular(return)`. (Where `return` stands for the return value of the method.) Yet the condition can also be on object state. For instance, we may have constraints such as:

```
Subrange(listTail,listRange)
```

This means that the state of the return value and overall object after execution of the method `listTail` is also a valid state after `listRange`: the former method produces a subset of the possible results and effects of the latter.

- *CanFollow(m1,m2)*: This constraint means that the postcondition of method *m1* implies the precondition of method *m2*. Execution of method *m2* is enabled by execution of *m1* in this sense, and it is a natural way to capture temporal sequencing rules, e.g.,:

```
CanFollow(open,write)
```

Such specifications are common and are often expressed in the form of state machines. Note that the above constraint means that "an `open` can be followed by a `write`", not "a `write` has to be preceded by an `open`". (The latter is inexpressible, since our vocabulary is only concerned with the relationship between conditions at two program points, but there is nothing to prohibit other program points from establishing the same conditions.)

- *Follows(m1,m2)*: This constraint means that the precondition of method *m2* implies the postcondition of method *m1*. Informally, this means that the only states that *m2* can start from are states that *m1 may* result in. *m1* does not guarantee such a state, though: *m1*'s postcondition is necessary but not sufficient for preparing to execute *m2*. In practice, this constraint prevents the postcondition of *m1* from being more specific than necessary. For instance, we may have:

```
Follows(add,remove)
```

This does not signify that `add` needs to execute before `remove` (a different method may be reaching the same states as `add`). But it means that if we are in a state where the last operation cannot have been an `add` (e.g., an empty structure), then we cannot `remove`.

- *Concord(m1,m2)*: *Concord* means that the postcondition of method *m1* implies that of method *m2* but only for the values that satisfy the preconditions of both *m1* and *m2* (or, put another way, only for the values for which the precondition of *m1* implies that of *m2*). This is a valuable constraint for methods that specialize other methods. For instance, there can be a fully general matrix multiply routine, a more efficient one for when the first argument is an upper triangular matrix, one for when the first argument is a diagonal matrix, etc. The operations' invariants are linked with a *Concord* constraint:

```
Concord(triangularMultiply, matrixMultiply)
```

Thus, the postcondition of the specialized operation (`triangularMultiply`) should imply that of the more general operation (indeed, in this case the conditions should be equivalent) for all their common inputs. Nevertheless, the general operation, `matrixMultiply`, is applicable in more cases and, thus, has a weaker precondition than the specialized one. For inputs valid for `matrixMultiply` but not `triangularMultiply`, the *Concord* constraint imposes no restriction.

2.3 Meaning and Completeness of Second-Order Constraints

We intend for second-order constraints to be devices for expressing relationships between program elements at a high level. This means that the precise interpretation of the constraints will depend on the specifics of the invariant inference system we use and on its capabilities. As a guideline, we present the intended meaning of five of our constraints in Figure 1, column *Pre/Post*. (We omit the 'OnlyCareAbout' predicates as they are straightforward.) In that column we summarize the meanings of each constraint as logical relations over the preconditions Pre_m and postconditions $Post_m$ of all involved methods.

We assume in this presentation that all preconditions use the same names for parameters in the same position, and all postconditions use the same names for their return value. In other words, $Pre_{m_1} \implies Pre_{m_2}$ means that any constraints that Pre_{m_1} places on the first (second, third, ...) parameter of m_1, Pre_{m_2} also places on the first (second, third, ...) parameter of m_2.

As we can see, the first four predicates capture all possible 'straightforward' implications between the preconditions and postconditions of two methods. Predicate *Concord* meanwhile captures a more complex but practically important second-order constraint.

We will return to this figure later, when we describe our implementation.

2.4 Discussion

Second-order constraints can serve as documentation of program behavior in much the same way that first-order invariants can (i.e., for program understanding purposes). Furthermore, second-order constraints can offer external knowledge that helps steer invariant inference in the right direction. Certainly one reason to do this is to avoid erroneous invariants: second-order constraints serve to cross-check invariants and thus enhance the dynamic opportunities to invalidate them, even with a limited number of observations. Another important use of constraints, however, is in deriving more *relevant* invariants. Clearly, a simple use of *OnlyCareAbout...* can prevent some irrelevant invariants. Greater benefit can be obtained by more abstract constraints, however. Consider, for example, a constraint on the constructor of a class `C` and on one of `C`'s methods, `m`. If we have the constraint *Follows*(`C`, `m`) then we have a hint that the constructor establishes part of the precondition for `m`. Without the constraint, we may observe the constructor's execution and derive for a certain field `fld` of the class the postcondition "`this.fld == 100`". This inference would certainly be reasonable, if, whenever the constructor is executed, `fld` is assigned the value 100. Yet

[1] Our implementation provides some default restrictions on which variables are considered when second-order constraints are used, so that common cases are covered without explicit *OnlyCareAbout...* annotations. Namely, when two methods are linked by a second-order constraint, their invariants are defined over "matching" arguments and fields. Roughly, argument variables match if they are in the same position and the methods have the same arity, and class fields match if they have the same name and type.

Constraint	Pre/Post	Dataflow
Subdomain(m_1, m_2)	$\mathit{Pre}_{m_1} \implies \mathit{Pre}_{m_2}$	$\mathsf{in}_t(m_1) \subseteq \mathsf{in}_t(m_2)$
Subrange(m_1, m_2)	$\mathit{Post}_{m_1} \implies \mathit{Post}_{m_2}$	$\mathsf{out}_t(m_1) \subseteq \mathsf{out}_t(m_2)$
CanFollow(m_1, m_2)	$\mathit{Post}_{m_1} \implies \mathit{Pre}_{m_2}$	$\mathsf{out}_t(m_1) \subseteq \mathsf{in}_t(m_2)$
Follows(m_1, m_2)	$\mathit{Pre}_{m_2} \implies \mathit{Post}_{m_1}$	$\mathsf{in}_t(m_2) \subseteq \mathsf{out}_t(m_1)$
Concord(m_1, m_2)	$\mathit{Pre}_{m_1} \wedge \mathit{Pre}_{m_2} \implies (\mathit{Post}_{m_1} \implies \mathit{Post}_{m_2})$	$\mathit{Pre}_{m_1}(t) \wedge \mathit{Pre}_{m_2}(t) \implies \mathsf{out}_t(m_1) \subseteq \mathsf{out}_t(m_2)$

Figure 1: Meanings of second-order constraints.

the invariant could be overly specific. Given that we know that the constructor's postcondition is implied by `m`'s precondition, we can observe all the different circumstances when `m` is called (not necessarily right after construction). This may yield the broader precondition "`this.fld > 0`" for `m`. Because of the *Follows* constraint, the constructor needs to also register all the same values and, hence, changes its postcondition to "`this.fld > 0`" instead of the more specific "`this.fld == 100`". The generality can mean that the new postcondition is more meaningful from a program understanding standpoint. The specific value 100 may be just an artifact of the specific test cases used (i.e., the postcondition "`this.fld == 100`" may be erroneously over-specific) but, even if it is correct, it may be the result of an arbitrary technicality or a detail likely to change in the next version of the program. Certainly, the *Follows* constraint steers the invariant inference to the kind of invariant the user wants to see in this case: the condition that (partly) enables `m` to run.

It is worth noting that the principle of *behavioral subtyping* [18] can be viewed as a combination of constraints from our vocabulary. Informally, behavioral subtyping is the requirement that an overriding method should be usable everywhere the method it overrides can be used. This is a common concept, employed also in program analyzers (e.g., ESC/Java2 [3]) and design methodologies (e.g., "subcontracting" in Design by Contract [23, p.576]). When a method `C.m` overrides another, `S.m`, the behavioral subtyping constraints consist of a combination of *Subdomain*(`S.m`, `C.m`) and *Concord*(`C.m`, `S.m`). (Implicitly there are also *OnlyCareAboutVariable* and *OnlyCareAboutField* constraints that restrict both the preconditions and postconditions to be over parameters of method `m` and member variables of the superclass `S`.) In past work [4] we discussed how consistent behavioral subtyping can be supported in the context of dynamic invariant inference systems, and similar ideas have informed the present work.

3. DESIGN AND IMPLEMENTATION

We next discuss the crucial design questions regarding a second-order constraints facility, as well as our own implementation decisions. Before this, however, we introduce the concept of monotonicity for a dynamic invariant inference system. This concept plays a central role in our discussion and understanding.

3.1 Background: Monotonicity

An important consideration for our later implementation arguments is whether our invariant inference system is (mostly) *monotonic*: If supplied more values to observe, the conditions it will output will be logically weaker (i.e., broader). Note that monotonicity applies to the logical predicates and not to the number of invariants. There may be fewer invariants produced when more values are observed, but these will be implied by the invariants produced for any subset of the values.

Most dynamic invariant inference tools are in principle monotonic. Tools like DySy [5] and Krystal [17] use a combination of dynamic and symbolic execution. The invariants are computed from the boolean conditions that occur inside the program text, when these are symbolically evaluated. The more executions are observed, the more the invariants of a program point are weakened by the addition of extra symbolic predicates (in a disjunction). Similarly, tools like Daikon or DIDUCE have invariant templates that are (multiply) instantiated to produce concrete candidate invariants. The candidate invariants are conceptually organized in hierarchies from more to less specific. Values observed during execution are used to falsify invariants and remove them from the candidate set. The *most specific* non-falsified candidate invariant of each hierarchy is a reported invariant. For instance, if two candidate invariants "`x == 0`" and "`x >= 0`" are satisfied by all observed values of `x`, then the former will be reported, as it is more specific. This is a monotonic approach: It means that as candidate invariants are falsified, they are replaced by more general conditions.

Nevertheless, invariant detection tools often include non-monotonicities in their inference logic. For instance, Daikon uses a *confidence* threshold for invariants: An invariant is not output even when consistent with all observations, if the number of pertinent observations is small. Similarly, Daikon treats some boundary values and invariant forms specially. For instance, if the maximum value for a variable is -2, Daikon will not output an invariant of the form `x <= -2`, but if it additionally observes the value 0 for the variable, it will output `x <= 0` (an invariant that would have been true even without observing the 0 value, but would not have been reported). Furthermore, Daikon produces the weakest predicate (*true*) when no invariants can be inferred. These features entail non-monotonicity: Observing more inputs will produce a stricter invariant.

In general, some non-monotonicity is unavoidable in practically useful dynamic invariant inference tools, even if just in corner cases. For instance, if a tool is strictly monotonic, then it has to infer a precondition and postcondition of *false* for every method that happens to not be exercised by the test suite. (This is the only logical condition that is guaranteed to be weakened—as monotonicity prescribes—when we add an execution that does exercise the method.) Computing invariants of *false* is a sound approach (accurately captures observations) but largely useless. Any further use of the produced invariants cannot employ the method in any way, as it can never establish the preconditions for calling it or know anything about its results. Effectively, monotonic invariant inference would have to report that a method is forever unusable if it was not observed to be used in the test run. This and other similar examples in practically significant cases preclude the use of strictly monotonic dynamic invariant inference.

3.2 Implementation

We implemented second-order constraints in the Daikon system. (Daikon is not only the mainstream option for dynamic invariant inference, but also convenient for engineering purposes. The DySy tool [5], on which we have worked in the past, was another interesting prospect, but requires C#/.NET and a specialized, closed-source, dynamic-symbolic execution infrastructure.) There are two aspects to the implementation. First, we infer second-order automatically through dynamic observations of program behavior. Second, we allow treating second-order constraints as "ground truth" and use them to influence regular invariant inference.

***Dynamically inferring second-order constraints*:** The key construct in most of our second-order constraint is an implication between pre/post conditions *P* and *Q* at two different program points (with each one being either the entry or the exit of a program procedure). Our strategy for proving the second-order constraint $P \implies Q$ is to first use Daikon (Rev. 2eded4baf181) to produce *P* and *Q* and then employ the Simplify theorem prover to check whether *P* can imply *Q*. The full implication check may fail, even if the implication holds in principle. One reason for this is that an inadequate test suite may lead Daikon to produce a slightly inaccurate *P* or *Q*. Furthermore, the theorem prover may be unable to decide whether an implication is satisfiable or not. Therefore, a more realistic check, since *P* and *Q* are expressed as conjunctions of invariants, is to see if a particularly large number of invariants in *Q* are subsumed by invariants of *P* (i.e., implied by the full conjunction *P*).

In order to find whether a sufficiently large number of implications between *P* and the constituent invariants of *Q* are true, we define the *success rate* (*SA*) of the implication as $SA = \frac{N}{M}$, where *N* represents the number of invariants in *Q* that can be implied by *P* and *M* denotes the number of invariants of *Q*.

The success rate on its own is often not enough for comparing the likelihood of a second order constraint being true. In our definition of success rate, all invariants are treated equally, so in a case with lots of junk invariants in *P* but not in *Q* that ratio may get overwhelmed by the junk. Daikon uses a *confidence* metric to approximate the probability that an invariant cannot hold by chance alone. If we sum up the confidence values for all invariant implications between *P* and each constituent invariant of *Q*, then the weight of the junk invariants in the ratio will decrease. Thus, we extend the definition of confidence to second-order constraints in order to approximately measure the probability that a second-order constraint could not hold by chance alone. For regular invariants, Daikon defines the confidence of an implication relation ($Inv1 \implies Inv2$) to be the product of the confidence of the guard (*Inv1*) and the confidence of the consequent (*Inv2*). Similarly, the second-order constraint confidence of $P \implies Q$ can be defined as the mean of the confidences of the implication relations between *P* and each invariant of *Q* that is implied (according to Simplify). Representing the confidences of invariants in *P* by pc_1, pc_2, ..., pc_Z, and denoting the confidences of invariants of *Q* that can be implied by *P* by qc_1, qc_2, ..., qc_N, we calculate the second-order constraint confidence, *MA*, as $MA = \frac{pc_1 \cdot pc_2 \cdot \ldots \cdot pc_Z \cdot (qc_1 + qc_2 + \ldots + qc_N)}{N}$.

We filter out second-order constraints whose success rate is below a threshold (by default 0.75) and rank the unfiltered second-order constraints according to confidence. It is meaningful for the user to change the success rate threshold, possibly based on the accuracy of invariants produced by Daikon for the program at hand. The fewer redundant and irrelevant invariants a program has, the higher the program's success rate filter can be.

The time complexity of our algorithm is quadratic over the number of methods. We consider all combinations of program points inside a class as candidate second-order constraints. If there are n methods in a class, there are $2n$ program points, i.e., $2n(2n-1)$ potential second-order constraints. To verify a candidate second-order constraint, we invoke the Simplify prover on implications between invariants. Removing irrelevant and redundant invariants before applying our tool can help reduce theorem proving time. (Standard avenues for doing so with Daikon include using more representative test suites and employing the DynComp tool [12] for more accurate invariants.)

We use a few heuristics to improve the accuracy of derived second-order constraints. One heuristic is to normalize parameters: a parameter's name is based on its position in the parameter list. For example, if we examine the second-order constraint *Subdomain*(`foo(int i, int j)`, `bar(int x, int y)`) then we do not use the names *i* and *j* when we compare to `bar(int x, int y)`. Instead, we substitute *arg1* for *i* and *x*, and *arg2* for *j* and *y*. We implement this normalization via regular expression transformation on the data trace file generated by Chicory (the Java instrumenter inside Daikon). In addition, if a variable only exists in the consequent-side program point of a second-order constraint, it often gets in the way of verification for the second-order constraint. We ignore invariants that contain such variables in the consequent-side program point. Another heuristic is that invariants over variables that are likely to be meaningless for cross-method comparisons are ignored. For instance, in the case of *Follows*(`bar`, `foo`) and *Subrange*(`foo`, `bar`), we ignore invariants over the *orig(x)* variable (which refers to the value of variable x upon entry to a procedure) at the exit of `bar`; for *Subrange*(`foo`, `bar`) and *CanFollow*(`foo`, `bar`), we ignore procedure parameters at both the entry and exit point of `bar`; for *Follows*(`bar`, `foo`), we ignore the return variable at the exit of `bar`.

***Using second-order constraints for better invariants*:** We extended Daikon with an annotation mechanism for second-order constraints. Users can pass a separate configuration file with second-order constraints to Daikon. This changes the Daikon processing of the low-level observations and allows refining the inferred invariants without having to re-run any test suites.

The main element of our approach is that when the precondition (or postcondition) of a method *m1* is constrained to imply the precondition (resp. postcondition) of another method *m2* (as shown in our earlier Figure 1, column *Dataflow*), we propagate the values observed at entry (resp. exit) of *m1* to *m2*. This ensures that producing *m2*'s invariants takes into account all the behavior observed for *m1*. Note that *m2* need not be executed during invariant inference—the conditions observed/established for *m1* are simply registered as if *m2* had also observed/established them. These observations are suitably adapted to be over common variables, as dictated by *OnlyCareAbout...* constraints (including implicit assumptions, as mentioned in Footnote 1).

For the first four second-order constraints of Figure 1 the implementation of the above value propagation is straightforward, as it can leverage existing Daikon facilities. Specifically, the implementation extensively hijacks the Daikon *dataflow hierarchy* mechanism. This Daikon internal facility propagates primitive invariants between different program points (a program point is a line of code or the entry or exit from a method). The machinery is therefore quite suitable (with some minor adjustment to suppress filtering in some cases) for implementing our value propagation.

However, the *Concord* second-order constraint is more complex: we only want to flow primitive invariants from $\mathsf{out}_t(m_1)$ to $\mathsf{out}_t(m_2)$ if at execution point t the preconditions of both m_1 and m_2 hold. However, at this point in Daikon's execution we do not yet know what the preconditions of m_1 and m_2 should be, since we have not even seen all primitive invariants yet. We thus evaluate *Concord* constraints in two passes. The first pass stores all data from $\mathsf{in}_t(m_1)$, $\mathsf{in}_t(m_2)$, and $\mathsf{out}_t(m_1)$. At the end of this pass we ask Daikon to compute the preconditions P to m_1 and m_2 using Daikon's usual heuristics (and any other second-order constraints). In the second pass, we now use these preconditions on the data from $\mathsf{in}_t(m_1)$ and $\mathsf{in}_t(m_2)$: $P(m_i, t)$ holds iff $\mathsf{in}_t(m_i)$ satisfies the precondition to m_i at point t. We thus iterate over t one more time to copy all primitive invariants from $\mathsf{out}_t(m_1)$ to $\mathsf{out}_t(m_2)$ and update Daikon's results as needed.

For example, assume that we are examining two methods int m1(int x) and int m2(int y). We make the following observations about m1:

m1: t	in_t(m1)	out_t(m1)
t_1	$\texttt{arg1} = -3$	$\texttt{return} = -1$
t_2	$\texttt{arg1} = 0$	$\texttt{return} = 0$
t_3	$\texttt{arg1} = 3$	$\texttt{return} = 1$
t_4	$\texttt{arg1} = 6$	$\texttt{return} = 2$

That is, the method is invoked at time t_1 with its first (and only) parameter x bound to -3, and returns -1, etc. From the table above, Daikon might plausibly infer the precondition $\mathit{Pre}_{\mathsf{m1}} \equiv \mathsf{true}$ and postcondition $\mathit{Post}_{\mathsf{m1}} \equiv \texttt{return} = \texttt{arg1}/3$.

Now assume that we make the following observations about m2:

m1: t	in_t(m2)	out_t(m2)
t_5	$\texttt{arg1} = 0$	$\texttt{return} = 0$
t_6	$\texttt{arg1} = 1$	$\texttt{return} = 0$
t_7	$\texttt{arg1} = 2$	$\texttt{return} = 0$

Here, Daikon sees no negative inputs and a constant output, so it might infer precondition $\mathit{Pre}_{\mathsf{m2}} \equiv \mathsf{arg1} \geq 0$ and postcondition $\mathit{Post}_{\mathsf{m2}} \equiv \mathsf{return} = 0$.

If the user intended m2 to be a specialized version of m1 for nonnegative integers, the precondition for m2 would be correct but the postcondition would be wrong. The user can address this by adding the constraint Concord(m1, m2). This constraint will make our system first compute preconditions and postconditions as above, then compute all the times t at which $\mathit{Pre}_{\mathsf{m1}}(t)$ agrees with $\mathit{Pre}_{\mathsf{m2}}(t)$. $\mathit{Pre}_{\mathsf{m1}}$ is always true, so we will only filter out t_1 via $\mathit{Pre}_{\mathsf{m2}}$. Consequently, Daikon will get to see all inputs and outputs from t_2, t_3 and t_4 in addition to the ones it was already considering for method m2. With this additional data, Daikon can no longer infer $\mathit{Post}_{\mathsf{m2}} \equiv \texttt{return} = 0$ but might instead conclude $\mathit{Post}_{\mathsf{m2}} \equiv \texttt{return} = \texttt{arg1}/3 \wedge \texttt{return} \geq 0$, which matches the user's intention.

3.3 Discussion

It is evident from the previous section that our two mechanisms (that of dynamically inferring second-order constraints and that of taking them into account when inferring invariants) operate differently. The former follows a static approach for checking invariant implication: the Simplify system is used as a symbolic prover. However, the mechanism of enforcing invariant implications (to produce different invariants by taking second-order constraints into account) eschews symbolic reasoning in favor of propagating more dynamic observations. Why do we not just take the conjunction of the produced invariants and declared second-order constraints, simplify it symbolically, and report it as the new produced invariants? The reason is that there is noise introduced when generalizing from observed executions to invariants (e.g., because the invariant patterns are uneven), and this carries over to the combined invariants. *It is better to generalize (i.e., compute invariants) from more observations than to generalize from fewer ones and then combine the generalizations symbolically.*

The above insight is evident in the case of non-monotonic invariant inferences, which are virtually unavoidable, as mentioned in Section 3.1. For instance, consider a *Subdomain(m1,m2)* constraint. If P is the precondition of *m1* and Q is the precondition of *m2*, then we can statically satisfy the constraint by considering the real precondition of *m2* to be $P \vee Q$, so that it is always implied by P. If, however, method *m1* is never called in our test suite, Daikon will infer *true* as its precondition. This will make *true* also be the precondition of *m2*, even though we have actual observations for that method! This result is counter-intuitive and so common in practice as to significantly reduce the value of produced invariants.

In contrast, in our chosen approach, a second-order constraint just causes more observations to register. These observations are then generalized using the same approach as the base inference process—i.e., just as if the system had really registered these observations. This is particularly beneficial in cases of non-monotonicity. Consider again our example of an *m1* and *m2* with preconditions P and Q, respectively, and a constraint *Subdomain(m1,m2)*. If *m2* observes the exact values that led to the inference of P, then these values, combined with the ones that led to the inference of Q, may induce higher confidence for an invariant more specific than either P or Q, which will now be reported (because of crossing a confidence threshold).

***Caveats*:** Note that our approach (of propagating observations from one program point to another) does not strictly guarantee that the dynamically inferred invariants satisfy the second-order constraints. Interestingly, *if the inference process is monotonic, correctness is guaranteed*: under monotonic invariant inference, taking into account the union of two sets of observations should produce a condition that is weaker than either individual condition. This observation argues for why our approach is expected to be correct: as long as there are enough observations, dynamic invariant inference is typically monotonic, as discussed in Section 3.1.

Additionally, any implementation of dynamic invariant inference under second-order constraints suffers from the possibility of a disconnect between observed executions and values reflected in an invariant. The source of the problem is that we are allowing an invariant to be influenced by values not really seen at that program point during execution. These values will be reflected in a reported invariant but will not be reflected in other dependent invariants. For instance, using second-order constraints we may infer a more general precondition, but not the corresponding postcondition. Thus, readers who consider the two invariants together may misinterpret their meaning even when the invariants are individually correct.

4. EVALUATION

There are two questions that our evaluation seeks to answer:

- Do second-order constraints aid the inference of better first-order invariants?

- Can correct second-order constraints be inferred dynamically?

The next two sections address these questions in order.

4.1 Impact of Second-Order Constraints

We explored the utility of second-order constraints in three case studies: one of a manageable, small example application with a relatively thorough test suite, and two of large, unfamiliar programs, with their actual test suites. In all studies, we first took existing classes and examined their APIs. We then added second-order constraints to manifest implicit relationships between API methods in the same class or in different classes. We ran a series of experiments to determine the effects that adding these constraints had on the observed pre- and postconditions reported by Daikon.

In these three case studies, second-order constraints were written by hand and their correctness was verified by inspection. This is a feasible approach in several settings since second-order constraints are much sparser/coarse-grained than first-order invariants: one needs to write few second-order constraints to affect a large number of first-order invariants.

```
public class StackAr {
    public boolean isEmpty();
    public boolean isFull();
    public void makeEmpty();
    public void push(Object);
    public void pop();
    public Object top();
    public Object topAndPop();
}
```

Figure 2: The array-based stack StackAr, shipped with Daikon

4.1.1 StackAr

Our first case study is `StackAr`, an array-based fixed-size stack implementation that ships with Daikon and is perhaps the most common Daikon benchmark and demonstration example. `StackAr` is interesting because it is the most controlled of our case studies (due to test suite coverage and small size).

Figure 2 lists the class and its methods. Methods `isEmpty` and `isFull` are straightforward. `makeEmpty` clears the stack. `push(o)` pushes element `o` to the top of the stack. `pop()` removes the top stack entry, but does not return a value. Instead, `top()` peeks at the top of the stack and returns the most recently pushed value; while `topAndPop()` returns the top of the stack before removing it. Operations `top` and `pop` raise an exception if the stack is empty, while `topAndPop` returns `null` in that case.

To explore our invariants, we examined this API and determined second-order constraints that we considered to be meaningful for this class. The process took one of the authors negligible time (less than a minute, although there was previous discussion of interesting invariants, hence the exact effort cost is unclear). We split these constraints into separate experiments and explored the effect they had on Daikon's invariant detection mechanism:

- Experiment 1 (**Ex1**): *Subdomain* on `topAndPop`, `top`, and `pop`. The operations `top`, `pop`, and `topAndPop` all require a nonempty stack, so we instructed the system to treat all of them as having identical subdomains.
- Experiment 2 (**Ex2**): Any *push* sets up the stack for a *top*, *pop*, or *topAndPop*. We experimented with setting up *CanFollow* relations between *push* and the three *top/pop* operations.

To ensure the highest-quality invariants, we ran these experiments with the DynComp tool [12] enabled.

Experiment 1: We instructed the system to treat all of `top`, `pop`, and `topAndPop` as having the same subdomains, using specifications such as

```
Subdomain(StackAr.topAndPop(), StackAr.pop())
Subdomain(StackAr.pop(), StackAr.top())
Subdomain(StackAr.top(), StackAr.topAndPop())
```

The above specification captures a circular subrange dependence, and hence equality: it specifies that all three operations should have effectively the same preconditions. There are other ways to express this equality: we experimented with reversing the circular dependencies and with establishing mutual *Subdomain* constraints between all three interesting pairs of the three operations (for a total of six constraints). We found these approaches to be equivalent.

The experiment results in the elimination of five spurious invariants from `pop`:

- '`this` has only one value'
- '`this.theArray` has only one value'
- '`size(this.theArray[]) == 100`'
- '`this.theArray[this.topOfStack] != null`'
- '`this.topOfStack < size(this.theArray[])-1`'

Also, "`this.topOfStack < size(this.theArray[])-1`" is replaced by the weaker (and correct) "`this.topOfStack <= size(this.theArray[])-1`". Our constraints further helped infer two new correct preconditions: one establishing an inequality between a stack's default capacity and the size of `this.theArray`, and one establishing that the top of the stack does not exceed the size of the internal array. As a result, the preconditions between the three methods were identical.

Experiment 2: For this experiment we specified that the `push` operation sets up a stack for using `top` and similar operations:

```
CanFollow(StackAr.push(Object), StackAr.top())
CanFollow(StackAr.push(Object), StackAr.pop())
CanFollow(StackAr.push(Object),
        StackAr.topAndPop())
```

This specification improved the inferred invariants, removing four incorrect preconditions (the same as for experiment #1, except for '`this` has only one value') and adding three invariants:

- One states that `topOfStack` cannot exceed the internal array size (as in experiment #1).
- One establishes an inequality between stack default capacity and array size (as in experiment #1).
- One establishes that the `topOfStack` is nonnegative.

Again the changes affected method '`pop`', while '`top`' and '`topAndPop`' remained unaffected, as they already had high test coverage.

In summary, the impact of second-order constraints on `StackAr`, under the standard test suite was overwhelmingly positive. All of the additional invariants were correct and most were insightful, while spurious invariants were eliminated.

4.1.2 Apache Commons Collections

Our second case study is the Apache Commons Collections library,[2] version 3.2.1. This library contains 356 classes, of which we used a total of 18 explicitly[3] for our experiments.

For our experiments, one of the authors (unfamiliar with the library) examined the above classes and constructed a specification (while consulting the API documentation) with a total of 27 second-order constraints, comprising 22 experiments. This process took approximately 3.5h. The author spent another 2h double-checking and fixing the invariants after specification. This amount of effort is in the noise level, compared to the development effort for a library of this size.

We then examined the utility of our specifications by running Daikon first with and then without our constraints. Since running the entire test suite with Daikon's instrumentation tool would have consumed harddisk space in excess of 10 GB (compressed), we instructed Daikon's instrumentation tool to only report invariants for the methods we were interested in. We only ran these experiments with DynComp disabled, as using the DynComp-instrumented JDK caused errors during execution.

We recorded the invariants of all affected methods and analyzed the generated differences. Figure 3 summarizes our results. We first list the affected class or classes, then the concrete second-order constraints we introduced (slightly compressed for space).

[2] `http://commons.apache.org/collections/`

[3] Some classes may use other classes from the library internally.

#	**Classes**	**second-order constraints**	**First-order invariants**			
			pre	**post**	**δpre**	**δpost**
1	ArrayStack	*CanFollow(push, peek)*	1	3	-1	
2	ArrayStack	*Subrange(peek, get)*	0	5		-2
3	ListUtils	*Subdomain(removeAll, retainAll)*	12	0	-2	
4	ListUtils	*Subdomain(sum, subtract)*	0	0		
5	CollectionUtils	*Subdomain(subtract, disjunction)*	8	0	+2	
6	CollectionUtils	*Subdomain(subtract, intersection)*	14	0		
7	TreeBidiMap	*Subdomain(containsKey, get)*	101	0	+6-7	
8	TreeBidiMap	*Subdomain(nextKey, previousKey)*	108	0		
9	UnmodifiableSortedBidiMap	*Subrange(tailMap, subMap)*	0	16		
10	TreeBidiMap	*Subdomain(remove, get),* *Subdomain(get, remove)*	99	0	+2-7	
11	TreeBidiMap	*Subdomain(getKey, removeValue),* *Subdomain(removeValue, getKey)*	93	0	+2-7	
12	DualTreeBidiMap	*Subrange(tailMap, subMap)*	0	22		
13	UnboundedFifoBuffer	*Follows(add, remove)*	11	34		+1-1
14	SetUniqueList	*Follows(addAll, remove)*	3	8		-1
15	AbstractBidiMapDecorator	*Subrange(getKey, removeValue)*	0	10		+3-1
16	ReverseListIterator	*Subrange(previousIndex, nextIndex),* *Subrange(nextIndex, previousIndex)*	0	20		+5-1
17	CursorableLinkedList	*CanFollow(addNode, removeNode)*	3	6		
18	CursorableLinkedList	*CanFollow(addNode, updateNode)*	5	6		
19	LinkedMap	*Subdomain(get, remove),* *Subrange(get, remove)*	4	6	+1-1	+1-1
20	ListOrderedMap	*Subrange(put, remove)*	0	16		-3
21	CompositeMap	*CanFollow(addComposited, removeComposited)*	8	19	+6-3	
22	CompositeCollection	*Concord(addComposited(Collection), addComposited(Collection, Collection))*	12	28		+2-2

Figure 3: Summary of our experiments on the Apache Commons Collections.

Next, we considered the invariants inferred by Daikon, restricted to invariants of the particular methods occurring in our constraints. For example, in experiment #1, we considered methods `push` and `peek` in class `ArrayStack` only. The final four columns summarize these invariants: first, we give the number of invariants in the absence of any second-order annotations, separated into pre- and postconditions (**pre** and **post**). Finally we give the differences over any preconditions (δ**pre**) and postconditions we observed (δ**post**). We use the notation $+x-y$ to indicate that we added x new invariants and removed y existing ones.

Qualitatively, the use of second-order constraints on the Apache Commons Collections was a clear win. All 35 invariants removed were false, to the best of our understanding. We added 26 invariants, of which our manual inspection found 25 to be true (i.e., expected to hold for all executions, not just the ones observed) and 1 to be false. The added invariants arise due to non-monotonicity. In experiment 16, for example, the augmented observations in the presence of second-order constraints enable two additional true invariants (i.e., a stronger inference) in the precondition of method `nextIndex`: `this.list != null` and `this.iterator != null`. The invariant "`orig(value) != null`" was incorrectly added to `AbstractBidiMapDecorator.removeValue(Object)` in experiment 15, where parameter `value` does not have to be non-null. Furthermore, we replaced 5 invariants with more general invariants. Consider, for example, experiment 13, with second-order constraint `Follows(add,remove)`. The postcondition of method `UnboundedFifoBuffer.add(Object)` originally contained invariants such as "`this.head one of {0, 1, 2}`". Such false invariants are replaced with a more insightful "`this.head >= 0`".

4.1.3 *AspectJ Compiler*

Our third case study is the AspectJ compiler. We followed the same approach as for the Apache Commons Collections, collecting invariants from unit tests and integration tests. Since AspectJ lacks detailed API documentation, one of the authors (unfamiliar with the library) directly inspected the source code of AspectJ and derived a total of 27 second-order constraints. The combined process of understanding the foreign code base and writing invariants cost the author approximately 10h. Again DynComp's instrumented JDK caused problems during execution, so we only tested with DynComp disabled.

We summarize our results in Figure 4. We removed 12 invariants. All of the 12 invariants were false. For instance, most "variable has only one value" and "variable is one of $\{\ldots\}$" invariants (largely due to limitations in the test suite) were removed or replaced by more accurate invariants. We also added 1 false invariant in experiment 16 (again, due to non-monotonicity) by assuming a variable is always less than a constant. Meanwhile, we added 12 true invariants and replaced 3 invariants with more general ones. For instance, without the second-order constraint `Subrange(getAjType, getDeclaringType)` in experiment 1, Daikon reports no invariants for the exit of the `getDeclaringType` method. Our `Subrange` constraint yields two new postconditions, "`return != null`" and "`return.getClass() == AjTypeImpl.class`", due to observations on `getAjType`.

4.2 Inferring Second-Order Constraints

We next evaluate the success of our dynamic process of inferring second-order constraints. Note that dynamically inferred second-order constraints are useful for many reasons:

- As documentation of program behavior, on their own, i.e., as deeper invariants than typical Daikon invariants.
- For finding bugs in manually stated second-order constraints.
- For offering the programmer a set of mostly-correct second-order constraints to choose from.

#	Classes	second-order constraints	First-order invariants			
			pre	post	δpre	δpost
1	AjTypeSystem, AdviceImpl	*Subrange(getAjType, getDeclaringType)*	0	9		+2
2	ProgramElement	*Subdomain(toSignatureString(boolean), toLabelString(boolean))*	76	0		
3	ProgramElement	*Subdomain(toLabelString(boolean), toSignatureString(boolean))*	76	0		
4	ProgramElement	*Subrange(toLabelString(), toLabelString(boolean))*	0	142		
5	ProgramElement	*Subdomain(getParent, getChildren)*	76	0		
6	ProgramElement	*Subrange(genModifiers(), getModifiers(int))*	0	80		
7	ProgramElement	*Subdomain(toLabelString(), toSignatureString())*	81	0	+1-5	
8	ProgramElement	*Subdomain(toSignatureString(), toLabelString())*	81	0		
9	ProgramElement	*Subdomain(getChildren, getParent)*	76	0	-1	
10	FieldSignatureImpl	*Subdomain(getDeclaringTypeName, getDeclaringType)*	0	0		
11	FieldSignatureImpl	*Subdomain(getDeclaringType, getDeclaringTypeName)*	0	0		
12	MethodSignatureImpl	*Subdomain(getField, getFieldType)*	0	0		
13	MethodSignatureImpl	*Subdomain(toShortString, toLongString)*	0	0		
14	MethodSignatureImpl	*Subdomain(getDeclaringType, getDeclaringTypeName)*	0	0		
15	SignatureImpl	*Subdomain(getDeclaringTypeName, getDeclaringType)*	66	0		
16	SignatureImpl	*Subdomain(getDeclaringType, getDeclaringTypeName)*	66	0	+4-4	
17	SignatureImpl	*Subdomain(toLongString, toShortString)*	71	0	+3	
18	SignatureImpl	*Subdomain(toShortString, toLongString)*	71	0	+3-1	
19	BcelWeaver	*CanFollow(prepareForWeave, weave(UnwovenClassFile, BcelObjectType))*	52	80	+3-4	
20	BcelWeaver	*CanFollow(prepareForWeave, weave(UnwovenClassFile, BcelObjectType, bool.))*	41	80		
21	BcelWeaver	*CanFollow(prepareForWeave, weaveAndNotify)*	54	80		
22	BcelWeaver	*CanFollow(prepareForWeave, weaveNormalTypeMungers)*	53	80		
23	BcelWeaver	*CanFollow(prepareForWeave, weaveParentTypeMungers)*	53	80		
24	BcelWeaver	*CanFollow(prepareForWeave, weave(IClassFileProvider))*	52	80		
25	BcelWeaver	*CanFollow(prepareForWeave, weaveParentsFor)*	57	80		
26	BcelWeaver	*CanFollow(prepareForWeave, weaveWithoutDump)*	51	80		

Figure 4: Summary of our experiments on the AspectJ Compiler.

Although the first benefit is very important, it is hard to quantify experimentally. Therefore, we focus on the second and third benefits, and specifically on evaluating the correctness of automatically inferred second-order constraints.

***Correctness of Inferred Constraints*:** We inspected the results of our automatic mechanism for inferring second-order constraints (Section 3.2) on four randomly selected classes from ACC and AspectJ (two each). One of the authors manually verified all of the generated second-order constraints. For this experiment, we considered a second-order constraint to be correct whenever it did not disagree with the implementation of the given class. Figure 5 lists our precision results. We note that overall precision exceeds 99%, suggesting that our confidence metric is highly effective at identifying low-quality constraint candidates.

Our five losses in `LstBuildConfigManager` were three *Follows* and two *Subrange* constraints involving methods with low-quality invariants. For example, four of the five constraints involved an `addListener` method for which Daikon had failed to observe the method's effect on the internal object state.

Although these constraints are true, they are not necessarily all interesting. Many of them just reflect implementation artifacts and would not arise if the methods in question had interesting invariants to begin with. For instance, for two methods that have a very small number of invariants in their preconditions, it is easy to find meaningless agreement—e.g., on the fact that their argument is never `null`.

For `SingletonMap`, we observed more than 800 proposed high-confidence constraints that claimed that most methods were in some relationship to each other. We found that `SingletonMap` is an immutable class, meaning that methods do not influence each other on subsequent calls—therefore many invariants remained the same across methods, resulting in second-order constraints being derived. (These include the two second-order constraints we manually wrote for `SingletonMap` but also many others of much less value.) This suggests the existence of other useful higher-order constraints beyond the catalog we have proposed herein; for `SingletonMap`, a meta-constraint *Immutable* would be the most concise way to express the properties that we observed.

In summary, we found that dynamic second-order constraint inference is highly effective at identifying high-quality sets of second-order constraints, even though we allow some margin of error over already-erroneous or imprecise axioms.

***Inferred vs. Manual Constraints*:** Our manually derived second-order constraints of the previous section were never intended as a full or ideal set, but as a set of constraints that take low effort to produce and have significant effect over first-order invariants. Still, it is instructive to compare them with automatically inferred constraints. For this purpose, we ran our inference mechanism on all classes that we had manually written second-order constraints for.

Indeed, our manual effort to produce constraints for ACC and the AspectJ compiler originally yielded 64 constraints and not just the 52 shown in Figures 3 and 4. 12 manually derived second-order invariants were removed exactly because their absence from the set of automatically inferred constraints caused us to re-inspect them and discover they were erroneous! This pattern is likely to also occur in practice when dealing with unfamiliar code (in fact, 8 of the 12 were in AspectJ, which lacks documentation). Note that in a scenario in which software authors write their own second-order constraints, such disagreements between hand-written and inferred second-order constraints would point to more serious issues; likely to poor test suites or to implementation bugs.

Of the 52 constraints in Figures 3 and 4 our automatic inference mechanism produced 37 and missed 15. On closer inspection we found that 6 of those missing hand-written second-order constraints are rejected (although true) due to low success rate. This is caused by "noise" invariants participating in the corresponding preconditions and postconditions, due to very few actual observa-

					Second-Order Constraints #		
Library	Class	methods #	Program points #	Daikon invariants #	total	correct	incorrect
ACC	AbstractMapBag	25	7	85	2	2	0
ACC	SingletonMap	34	54	635	806	806	0
AspectJ	Reflection	17	10	192	30	30	0
AspectJ	LstBuildConfigManager	18	23	778	112	107	5

Figure 5: Results of inferring second-order constraints. In the above, 'program points' lists only program points that our inference considers, i.e., `:::ENTER` and `:::EXIT` program points.

tions for the relevant methods. 7 more second-order constraints are missing due to having no data samples at all for the methods. The last 2 second-order constraints are missing since our current implementation does not support detecting the *Concord* constraint or constraints relating methods in two different classes.

Thus, overall the automatic inference facility produces quite high-quality results on its own, and is found to be a strong complement for manual derivation of second-order constraints.

5. RELATED WORK

There is a wealth of other work on invariant inference. We next selectively focus on some recent approaches that were not covered in the body of the paper (typically because they focus on static invariant inference techniques).

For reverse engineering, Gannod and Cheng [10] proposed to infer detailed specifications statically by computing the strongest postconditions. Nevertheless, pre/postconditions obtained from analyzing the implementation are usually too detailed to understand and too specific to support program evolution. Gannod and Cheng [11] addressed this deficiency by generalizing the inferred specification, for instance by deleting conjuncts, or adding disjuncts or implications. Their approach requires loop bounds and invariants, both of which must be added manually.

There has been some recent progress in inferring invariants using abstract interpretation. Logozzo [20, 21] infers loop invariants while inferring class invariants. The limitation of his approach are the available abstract domains; numerical domains are best studied. Resulting specifications are expressed in terms of fields of classes.

Flanagan and Leino [7] propose a lightweight verification-based tool, named Houdini, to statically infer ESC/Java [8] annotations from unannotated Java programs. Based on pre-set property patterns, Houdini conjectures a large number of possible annotations and then uses ESC/Java to verify or refute each of them. The ability of this approach is limited by the patterns used. In fact, only simple patterns are feasible, otherwise too many candidate annotations will be generated, and, consequently, it will take a long time for ESC/Java to verify complicated properties.

Taghdiri [25] uses a counterexample-guided refinement process to infer over-approximate specifications for procedures called in the function being verified. In contrast to our approach, Taghdiri aims to approximate the behaviors for the procedures within the caller's context instead of inferring specifications of the procedure.

Henkel and Diwan [15] have built a tool to dynamically discover algebraic specifications for interfaces of Java classes. Their specifications relate sequences of method invocations. The tool generates many terms as test cases from the class signature. The results of these tests are generalized to algebraic specifications. They use a second tool to dynamically compare their specifications against implementations by executing both simultaneously and comparing their behavior [16].

Much of the work on specification mining is targeted at inferring API protocols dynamically. Whaley et al. [26] create a finite state machine into which a transition from method A to method B is added if the post-condition of method A is not mutually exclusive with the pre-condition of method B. Meghani and Ernst [22] build upon Whaley's work by using Daikon to determine the likely pre/post-condition of each method. Other approaches use data mining techniques. For instance Ammons et al. [1] use a learner to infer nondeterministic state machines from traces; similarly, Yang and Evans [27] built Terracotta, a tool to generate regular patterns of method invocations from observed runs of the program. Li and Zhou [19] apply data mining in the source code to infer programming rules, i.e., usage of related methods and variables, and then detect potential bugs by locating the violation of these rules. Gabel and Su [9] dynamically infer and verify method call ordering constraints, and report the constraints only if they are violated. Beschastnikh et al. [2] use mined temporal invariants from logs to derive a refined finite state machine. Although not explicitly our goal, some of our second-order constraints can be thought of as a way to express temporal API protocols. For example, we can find the general has∗, next∗ type specification [9] by checking if *CanFollow(has∗(return==true),next∗)*, or that the postcondition of has∗ when has∗ returns true implies the precondition of next∗. It is interesting future work to see how to integrate existing techniques on specification mining with our approach to derive automata that describe correct behavior.

6. CONCLUSIONS

Second-order constraints can steer dynamic invariant inference to avoid erroneous invariants and to derive more relevant invariants while reducing noise. We have defined a vocabulary of second-order constraints and described how each of them encodes information that is typically known by programmers and useful to a dynamic invariant detector. We have taken an approach in which the second-order constraints control the propagation of the observations on which invariant detection is based. We have also extended the Daikon system so as to infer second-order constraints.

Overall, we consider second-order constraints to be a particularly promising idea not just as a meaningful documentation concept but also for improving the consistency and quality of dynamically inferred invariants—a major challenge in this area.

7. ACKNOWLEDGMENTS

We gratefully acknowledge funding by the European Union under a Marie Curie International Reintegration Grant and a European Research Council Starting/Consolidator grant; by the Greek Secretariat for Research and Technology under an Excellence (Aristeia) award; and by by the U.S. National Science Foundation under grants CCF-0917391 and CCF-0916569.

Our tools are available at:
`http://code.google.com/p/getmetainv/`
`http://code.google.com/p/usemetainv4daikon/`.

8. REFERENCES

[1] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 4–16, New York, NY, USA, 2002. ACM Press.

[2] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 267–277, New York, NY, USA, 2011. ACM.

[3] David R. Cok and Joseph R. Kiniry. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2. Technical Report NIII-R0413, Nijmegen Institute for Computing and Information Science, May 2004.

[4] Christoph Csallner and Yannis Smaragdakis. Dynamically discovering likely interface invariants. In *Proc. International Conference on Software Engineering (Emerging Results Track)*, pages 861–864, May 2006.

[5] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *International Conference on Software Engineering (ICSE)*, May 2008.

[6] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001.

[7] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In José Nuno Oliveira and Pamela Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity: International Symposium of Formal Methods Europe*, pages 500–517. Springer, March 2001.

[8] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 234–245, June 2002.

[9] Mark Gabel and Zhendong Su. Online inference and enforcement of temporal properties. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 15–24, New York, NY, USA, 2010. ACM.

[10] Gerald C. Gannod and Betty H. C. Cheng. Strongest postcondition semantics as the formal basis for reverse engineering. In *Proc. Second Working Conference on Reverse Engineering (WCRE)*, pages 188–197. IEEE, July 1995.

[11] Gerald C. Gannod and Betty H. C. Cheng. A specification matching based approach to reverse engineering. In *Proc. International Conference on Software Engineering*, pages 389–398. ACM, May 1999.

[12] Philip J. Guo, Jeff H. Perkins, Stephen McCamant, and Michael D. Ernst. Dynamic inference of abstract types. In *Proceedings of the 2006 international symposium on Software testing and analysis*, ISSTA '06, pages 255–265, New York, NY, USA, 2006. ACM.

[13] Samuel Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. In *Second Conference on Domain-Specific Languages (DSL)*, pages 39–52, 1999.

[14] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, May 2002.

[15] Johannes Henkel, Christoph Reichenbach, and Amer Diwan. Discovering documentation for java container classes. *IEEE Trans. Software Eng.*, 33(8):526–543, 2007.

[16] Johannes Henkel, Christoph Reichenbach, and Amer Diwan. Developing and Debugging Algebraic Specifications for Java classes. *ACM Trans. Softw. Eng. Methodol.*, 17(3), 2008.

[17] Yamini Kannan and Koushik Sen. Universal symbolic execution and its application to likely data structure invariant generation. In *ISSTA '08: Proceedings of the 2008 Int. Symposium on Software Testing and Analysis*, pages 283–294, New York, NY, USA, 2008. ACM.

[18] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR98-06y, Department of Computer Science, Iowa State University, June 1998.

[19] Zhenmin Li and Yuanyuan Zhou. PR-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proc. 13th International Symposium on Foundations of Software Engineering (FSE)*, pages 306–315. ACM, September 2005.

[20] Francesco Logozzo. Automatic inference of class invariants. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, number 2937 in LNCS. Springer-Verlag, January 2004.

[21] Francesco Logozzo. *Modular Static Analysis of Object-Oriented Languages*. PhD thesis, Ecole Polytechnique, June 2004.

[22] Samir V. Meghani and Michael D. Ernst. Determining legal method call sequences in object interfaces, May 2003.

[23] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, 2nd edition, 1997.

[24] Jeff H. Perkins and Michael D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *Proceedings of the ACM SIGSOFT 12th Symposium on the Foundations of Software Engineering (FSE 2004)*, pages 23–32, November 2004.

[25] Mana Taghdiri. Inferring specifications to detect errors in code. In *Proc. 19th IEEE International Conference on Automated Software Engineering (ASE)*, pages 144–153, September 2004.

[26] John Whaley, Michael C. Martin, and Monica S. Lam. Automatic extraction of object-oriented component interfaces. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, pages 218–228. ACM, July 2002.

[27] Jinlin Yang and David Evans. Dynamically inferring temporal properties. In *Proc. 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 23–28. ACM, June 2004.