State Space Discovery by Guided Dynamic Analysis

Nadya Kuzmina, John Paul, Ruben Gamboa, and James Caldwell [*]
University of Wyoming
P.O. Box 3315
Laramie, WY 82071-3315
{nadya, jpaul, ruben, jlc}@cs.uwyo.edu

ABSTRACT

Dynamic constraint inference techniques give precise results, but are usually limited to relatively few properties. This limitation often prohibits a dynamic constraint inference approach from expressing the essential behavior of a given program. This paper introduces the state space partitioning technique which applies the partitions implicit in a program's source code to effectively introduce several types of disjunctive program-specific properties into the language of dynamic constraint inference. Disjunctive properties considered by our approach include object invariants and transitions between abstract states of the target program.

The state space partitioning technique has been implemented in a tool called ContExt, which relies on Daikon for dynamic constraint inference tasks. We demonstrate that our approach specifies the essential behavior on five examples from different domains. In each case pure dynamic constraint inference fails to capture important constraints on the essential behavior of the program under analysis. The primary reason for this failure is the inexpressibility of a general and effectively computable disjunctive constraint in the language of dynamic constraint inference.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Class invariants—*automatic specification recovery*

Keywords

dynamic constraint inference, class invariants, disjunctive constraints

1. INTRODUCTION

Dynamic constraint inference techniques output *likely* constraints by examining program executions. *Likely* constraints are properties that hold on the examined program runs [24]. In practice many of these *likely* constraints coincide with a program's essential specification. Dynamic constraint inference techniques have been successfully applied to a variety of problems, such as automating theorem proving tasks [19, 20], verifying safety properties [23, 22, 30], generating and prioritizing test cases [28], program refactoring [15], and error detection [26, 25] among others. Aside from the fact that it is unsound, dynamic constraint inference typically suffers from two limitations.

The first one which is shared by most constraint inference techniques whether static or dynamic is that the types of properties considered must be fixed by a human operator prior to analysis. The second one which is typical of the dynamic approach alone is that it is not designed to exploit structured property spaces.

The first limitation often results in analysis with the same fixed set of *a priori* identified properties that do not take the logic of the program under analysis into account. A fixed set of properties or templates is justified for a task like fault detection [8] which may only require the discovery of a pre-defined set of fault-specific properties for its success. A fixed property set, however, presents serious limitations for tasks like comprehension which often require the discovery of unforeseen properties.

That the dynamic approach does not depend on structured property spaces can perhaps be seen to mitigate this first limitation: it is easier to add a new property if this can be done independently of what is already there. It also, however, makes the inference of disjunctive combinations of properties, something that is quite easy to do certain structured domains, prohibitively expensive. Furthermore, if new properties are introduced by templates as they are in Daikon [9] for instance, then due to its unrelatedness to the others each new template will perforce be instantiated independently of all the others. So, a cost accrues to introducing new templates that would be avoided if the analysis were capable of maintaining dependencies between certain property instances when they are assumed to exist.

This paper attempts to overcome both limitations at once by viewing the conditional statements in the source code of the program under analysis, which is assumed to be a class with protected data members in an object-oriented language, as guiding the automatic construction of property domains whose structure can be used efficiently to generate hypotheses to be checked against a set of execution traces. Primarily this paper concentrates on the tests that a class performs on its own internal variables from within the constructors and methods it defines. The technique that allows for the inference of constraints based on these tests is called *state space partitioning*.

[*]This material is based upon work supported by the National Science Foundation under Grant No. NSF CNS-0613919.

The state space partitioning technique automatically derives an abstract state space for the class under analysis. From this space the technique constructs three types of hypotheses: object invariants, method preconditions, and the transitions induced by each method on this space. The hypotheses are then checked against a set of execution traces involving instances of the class. The result of the technique is a set of disjunctive constraints on the abstract states of the class. As is characteristic of dynamic approaches, the results of the state space partitioning technique are precise, but unsound.

The state space partitioning technique has been implemented in a tool called ContExt. ContExt relies on a cursory static analysis to determine its property spaces and on Daikon for dynamic constraint inference tasks. We demonstrate the inferences that ContExt was able to perform on five simple but non-trivial examples presented in Section 3. On each example, Daikon fails to infer at least a part of the essential specification due to the inexpressibility of disjunctive constraints in its language.

The rest of this paper is organized as follows. Section 2 is devoted to the description of the state space partitioning technique. More specifically, Section 2.1 provides the motivation behind the technique. Section 2.2 presents a formal algorithm for our approach. Section 2.2.6 discusses the implementation of ContExt. Section 3 presents and compares constraints inferred by ContExt and Daikon on five simple but non-trivial examples. Section 4 provides an evaluation framework for constaint inference tools. Discussion of our approach is presented in Section 5. Related work is discussed in Section 6, and finally, Section 7 presents our conclusions.

2. STATE SPACE PARTITIONING

2.1 Motivation

Throughout this paper we are concerned with the task of automatically inferring constraints on the behavior of *correct* target programs via program analysis. The ultimate goal of this task is to recover a program specification from the program itself that closely approximates its essential specification, the essential specification being a declarative description of the program's essential behavior. In the best case, the task we have outlined will recover the complete essential specification of a given program, where we are really only limiting ourselves to classes in an object-oriented programming language.

The languages of current dynamic constraint detection techniques are often specified by a fixed grammar of universal properties [9, 30]. These universal property languages often provide a means to obtain a kind of rough initial approximation of the program under analysis, but they are often not sufficient to express more subtle facts that describe the particular logic of this program apart from most others. It would, furthermore, be computationally prohibitive to go after these facts in the same way as the universal ones.

Instead we offer a technique to *automatically specialize* the language of constraint detection to a particular program on a per-program basis. The key observation for our technique is that certain constructs from the source code can be mapped to assumptions about the target program, an idea that goes back to the advent of structured programming [6]. Such assumptions are then used to generate constraint hypotheses that are checked dynamically against a set of execution traces.

Our approach bears some similarity to that of Engler et. al. [8] who infer programmer "beliefs" (facts implied by code) and then cross-check them for contradictions with the purpose of discovering program errors. The similarity lies in that both Engler's approach and ours use source code to guess the programmer's intentions. Engler, however, is interested in the error-revealing beliefs of the form "pointer `p` is non-null" and "a call to `lock` must be followed by a call to `unlock`" in the context of large kernel programs, whereas we hypothesize constraints about what the state space of a particular class in an object oriented program looks like based on the way the programmer appears to partition this space with `if-then-else` statements.

$$\begin{array}{lcl} \texttt{if (x < 0) \{...\}} & & P_1 \equiv x < 0, \\ \texttt{else if (y > 0) \{...\}} & \Rightarrow & P_2 \equiv x \geq 0 \wedge y > 0, \\ \texttt{else \{...\}} & & P_3 \equiv x \geq 0 \wedge y \leq 0 \end{array}$$

Figure 1: An `if-then-else` statement and its corresponding partition

One way to view a class in an object-oriented system is as defining the space of all states attainable by instances of this class. The individual dimensions of this *state space* are indexed by the attributes declared in the class, and the space itself is contained within the Cartesian product of the domains of those attributes. A conditional statement appearing in the class can then be seen to distinguish object states into two or more subspaces that are used by the programmer to anticipate the distinctive behavior of objects whose states are thus categorized. Being disjoint from the others, each subspace serves as an abstract state representing the set of object states that it contains. Also, since the subspaces together exhaust the possibilities for the attribute dimensions they contain, they comprise an abstract state space.

In particular, each test in an `if-then-else`-statement exclusive of the preceding tests defines a partition on the values of attributes that participate in the tests of the statement. For example, the tests `x < 0` and `y > 0` in Figure 1 partition the state space $\{\langle x, y\rangle \mid -2^{31} \leq x, y < 2^{31}\}$ consisting of all possible pairs of `int` values for attributes `x` and `y` into three disjoint subspaces, or *states*, that are characterized by the additional facts that either $x < 0$, or $x \geq 0 \wedge y > 0$, or $x \geq 0 \wedge y \leq 0$. The questions that can be asked for each method in the class are then "Is the method always called by objects in a particular abstract state?" and "What transitions does the method induce on the abstract state space?". In terms of finite state machines (FSM), each conditional statement identifies the states of an FSM, and each method specifies transitions on these states just as the letter in an input alphabet would. Any nondeterminism in the FSM is described by a disjunction of abstract states.

Our technique operates on object-oriented programs and uses state space partitions to effectively introduce a number of different types of *disjunctive constraints* into the language of constraint detection. Disjunctive hypotheses based on state space partitions include constraints on the distinctive behavior of objects while they are in each abstract state, as well as constraints on the transitions between abstract states induced by class methods, and crucially, object invariants as will be seen in section 2.2.5. Since general disjunctive relations are expensive to compute, the universal properties of traditional dynamic analysis exclude disjunctive constraints.

2.1.1 The Calculator Example

```
public class CalcEngine {

// object invariant:
// (!adding || !subtracting)

//number which appears in the Calculator display
private int displayValue;
//store a running total
private int total;
//true if #'s pressed should overwrite display
private boolean newNumber;
//true if adding
private boolean adding;
//true if subtracting
private boolean subtracting;

/* preconditions:
 * P1 || P2, Q1 || Q2 || Q3
 * post-conditions:
 * orig(P1) ==> P2, orig(P2) ==> P2
 * orig(Q1) ==> Q1, orig(Q2) ==> Q2
 * orig(Q3) ==> Q3
 * orig(P1) <==> (displayValue == orig(number))
 * orig(P2) ==>
 * (displayValue == 10*orig(displayValue)+orig(number))
 */
public void numberPressed(int number) {
if (newNumber)
    displayValue = number;
else
    displayValue = displayValue * 10 + number;
newNumber = false;
}

public void equals() {
if (adding)
    displayValue = displayValue + total;
else if (subtracting)
    displayValue = total - displayValue;
  ...
}

/* preconditions:
 * P1 || P2, Q3
 * post-conditions:
 * orig(P1) ==> P1, orig(P2) ==> P1
 * orig(Q3) ==> Q3
 */
public void clear() { ... }

}
```

Figure 2: Calculator Example with two partitions

The `CalcEngine` class in Figure 2 represents a state-based calculator. The values of the `newNumber`, `adding`, and `subtracting` attributes participate in the state of a `CalcEngine` object and determine the action taken when a button on the calculator's keyboard is pressed. For example, when a number button is pressed, the `numberPressed` method is called. The behavior of the `numberPressed` method is determined by the `newNumber` value. If `newNumber` is `true`, then `displayValue` is assigned the number that was pressed; if `newNumber` is `false` then `displayValue` is set to `displayValue * 10 + number`.

Our technique forms two state spaces for the `CalcEngine` objects based on the tests of the conditional statements in the source code of the class. The first space $\mathbb{P}_1$ is derived from the `if`-statement in the body of the `numberPressed` method and consists of two abstract states, one called P_1 where `newNumber` is `true` and one called P_2 where `newNumber` is `false`. The second space $\mathbb{P}_2$ originates from the `if`-statement in the body of the `equals` method and consists of three abstract states Q_1, Q_2, and Q_3 defined by the predicates `adding`, $\neg$`adding` $\wedge$ `subtracting`, and $\neg$`adding` $\wedge \neg$`subtracting` respectively.

The constraints automatically inferred by our technique are presented on Figure 2. The precondition on the `numberPressed` method indicates that this method was called by `CalcEngine` objects in every abstract state of $\mathbb{P}_1$ and $\mathbb{P}_2$. The precondition on `clear`, however, suggests that this method was only invoked by objects in P_1, P_2, or Q_3. Preconditions reveal the "use-cases" of each method observed over a set of execution traces.

Post-conditions reflect the state transitions induced by a method, if any, by relating an initial abstract state observed at the precondition to the disjunction of abstract states that were observed at the post-condition of this method. For example the postconditions for the `numberPressed` method reveal that this method performs a transition from any initial $\mathbb{P}_1$-state into $P2$ and serves as identity function on the $\mathbb{P}_2$ states.

Our technique also automatically infers the object invariant $(\neg adding \vee \neg subtracting)$ for the `CalcEngine` class. This object invariant is an essential constraint which says that `adding` and `subtracting` are mutually exclusive in all `CalcEngine` instances.

2.2 Our Approach

Naively it appears that the computation of the disjunctive postconditions in the Calculator example would require us to check a number of state combinations exponential in the size of the state space for each possible transition. This section describes an algorithm that by exploiting the structure of the abstract state spaces must only consider a linear number of constraints per transition.

2.2.1 Test Conditions to Partitions

Every `if-then-else` statement defines a sequence of boolean expressions consisting of the test expressions mentioned by the statement in the order in which they appear in the statement. One arrives at a disjoint partition of the state spaces of the program variables involved in expressing these tests by making the semantics of the relative position of each test in this sequence explicit.

This is accomplished by conjoining each test with the negations of all the tests preceding it in the sequence and by adding an explicit *else*-partition that is the combined negation of all the tests in the sequence. The result is a set of logically disjoint formulas whose disjunction is a tautology. The cardinality of this set is one greater than the length of the original sequence.

For example, consider an `if-then-else` statement of the form

```
if (cond1) { ... }
else if (cond2) { ... }
else { ... }
```

The disjoint partition of the state space of the variables involved in $cond_1$ and $cond_2$ is constructed as follows: $P_1 = cond_1$, $P_2 = cond_2 \wedge \neg cond_1$, and $P_3 = \neg cond_2 \wedge \neg cond_1$. Each symbol P_1, P_2, or P_3 is used to denote both a logical formula and the respective subspace or *state* induced by this formula within the overall state space.

The symbol $\mathbb{P}$ will be used to denote an arbitrary partition whose

$\delta(m, \mathbb{P})$	1	2	$\dots$	n
m	$J(1, m, \mathbb{P})$	$J(2, m, \mathbb{P})$	$\dots$	$J(n, m, \mathbb{P})$

Figure 3: Transition relation computed for method m and partition $\mathbb{P} = \{P_1, P_2, \dots, P_n\}$

formulas are indexed by the set $I_{\mathbb{P}} = \{1, \dots, |\mathbb{P}|\}$ of the first $|\mathbb{P}|$ positive integers in such a way that *cond*$_i$ refers to the ith test for each $i < |\mathbb{P}|$. Thus, P_1 is identical with *cond*$_1$, and the final formula, when $i = |\mathbb{P}|$, always coincides with the *else*-formula. The simplest partitions are, therefore, those derived from single *if-then-else* statements and are indexed by the set $\{1, 2\}$.

It will also prove convenient to be able to refer to the method containing the conditional statement from which $\mathbb{P}$ is derived, and this will be denoted by $m(\mathbb{P})$.

If each *cond*$_i$ can be expressed in terms of instance variables and constants alone then $\mathbb{P}$ is said to be *class* scoped. If some *cond*$_i$ contains additional mention of a formal parameter to $m(\mathbb{P})$ then $\mathbb{P}$ is said to be an $m(\mathbb{P})$ scoped *input* partition. The notion of partition scope makes it possible to distinguish those partitions that can be evaluated anywhere in the class and are hence class scoped from those that can only be evaluated within the context of a specific method and are hence method scoped.

2.2.2 *Computing the Transition Relation*

Once the partitions and their respective scopes have been extracted from a class by a precursory static analysis, it is possible to approximate the transition relation $\delta(m, \mathbb{P})$ induced by method a m with respect to any class scoped partition $\mathbb{P}$.

To do this we approximate for every $i \in I_{\mathbb{P}}$ the smallest subset $J(i, m, \mathbb{P}) \subseteq I_{\mathbb{P}}$ such that the disjunction

$$\bigvee_{j \in J(i,m,\mathbb{P})} P_j$$

holds upon exit of method m given that P_i holds upon entry of m. The resulting transition relation is shown in Figure 3. A transition from P_i to P_j is said to be possible under m if $j \in J(i, m, \mathbb{P})$.

Stated in terms of Hoare triples, each $J(i, m, \mathbb{P})$ is the smallest subset such that

$$\{P_i\}\, m \left\{ \bigvee_{j \in J(i,m,\mathbb{P})} P_j \right\}$$

is a valid Hoare triple.

This minimum is well-defined: $I_{\mathbb{P}}$ itself satisfies the above triple since the logical disjunction over the entire partition is *true*, and as the following argument shows next, if $J(i, m, \mathbb{P})$ and $J'(i, m, \mathbb{P})$ both satisfy the triple so does their intersection.

If each disjunction individually satisfies the triple, then so does the conjunction of both

$$\{P_i\}\, m \left\{ \left(\bigvee_{j \in J(i,m,\mathbb{P})} P_j \right) \wedge \left(\bigvee_{j' \in J'(i,m,\mathbb{P})} P_{j'} \right) \right\}.$$

Redistributing yields the disjunction over all possible combinations $P_j \wedge P_{j'}$. Since P_j and $P_{j'}$ are, however, logically disjoint unless $j = j'$, this goes to show that only those indexes contained in both $J(i, m, \mathbb{P})$ and $J'(i, m, \mathbb{P})$ are the relevant part of each set.

The approach taken in this paper is to use a dynamic invariant detector to approximate each $J(i, m, \mathbb{P})$ with a potentially *smaller* set. The basic algorithm for a single partition $\mathbb{P}$ whose scope includes method m is given next.

1. For each $i \in I_{\mathbb{P}}$, initialize local variable $J^c(i, m, \mathbb{P})$ to $I_{\mathbb{P}}$.

2. Perform dynamic analysis, and whenever a counterexample of
$$\{P_i\}m\{\neg P_j\}$$
is exhibited for some $i \in I_{\mathbb{P}}$ and $j \in J^c(i, m, \mathbb{P})$, remove j from $J^c(i, m, \mathbb{P})$.

3. Approximate $J(i, m, \mathbb{P})$ with the set difference $J^*(i, m, \mathbb{P}) = I_{\mathbb{P}} - J^c(i, m, \mathbb{P})$.

The algorithm relies on the fact that each $\mathbb{P}$ is a valid logical partition. In particular, it leverages the fact that for any two complementary subsets J and J^c such that $J^c \cap J = \emptyset$ and $J^c \cup J = I_{\mathbb{P}}$

$$\bigvee_{j \in J} P_j = \bigwedge_{j \in J^c} \neg P_j.$$

Thus, step 2 of the algorithm effectively constructs the largest conjunction of negated partition formulas that is consistent with the observed runs over a particular dynamic analysis, and step 3 merely coverts this into the equivalent smallest disjunction. It is this last possibility that allows us to avoid having to work directly with the $2^{|\mathbb{P}|}$ disjunctions that may be the outcome of a single transition. Instead the algorithm must only consider $|\mathbb{P}|$ constraints per transition.

Because dynamic analysis is perfectly precise, any $\{P_i\}m\{\neg P_j\}$ eliminated in step 2 is a truly impossible Hoare triple. However, because dynamic analysis is unsound it may not eliminate all of the impossible triples. Thus, the approximation computed by the algorithm is related to the actual solution in the following way: $J^*(i, m, \mathbb{P}) \subseteq J(i, m, \mathbb{P})$ for each $i \in I_{\mathbb{P}}$. Seen from a logical point of view, the disjunctions indexed by the J^*s are potentially *stronger*, in the sense of entailment, than those indexed by the Js. The inequality therefore exemplifies the unsoundness that is associated with the dynamic approach in general.

2.2.3 *Refining the Transition Relation*

The discussion so far has focused on a transition relation $\delta(m, \mathbb{P})$ computed with respect to a class scoped partition $\mathbb{P}$. It is now shown how to refine this relation in the presence of any m scoped input partition $\mathbb{I}$.

Instead of computing only one set $J(i, m, \mathbb{P})$ for every $i \in I_{\mathbb{P}}$ we now compute $|I_{\mathbb{I}}|$ separate sets $J_k(i, m, \mathbb{P})$, one for every for every $Q_k \in \mathbb{I}$. The interpretation of $J_k(i, m, \mathbb{P})$ is that it is the smallest subset of $I_{\mathbb{P}}$ such that

$$\{Q_k \wedge P_i\}\, m \left\{ \bigvee_{j \in J_k(i,m,\mathbb{P})} P_j \right\}$$

$\delta(m, \mathbb{P}, \mathbb{I})$	1	2	$\ldots$	n
$m, 1$	$J_1(1, m, \mathbb{P})$	$J_1(2, m, \mathbb{P})$	$\ldots$	$J_1(n, m, \mathbb{P})$
$m, 2$	$J_2(1, m, \mathbb{P})$	$J_2(2, m, \mathbb{P})$	$\ldots$	$J_2(n, m, \mathbb{P})$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
m, l	$J_l(1, m, \mathbb{P})$	$J_l(2, m, \mathbb{P})$	$\ldots$	$J_l(n, m, \mathbb{P})$

Figure 4: Transition relation as refined by the input partition $\mathbb{I} = \{Q_1, Q_2, \ldots, Q_l\}$

is a valid Hoare triple.

The algorithm for approximating these sets is essentially the same as before, and the resulting transition relation is depicted in Figure 4. A transition from P_i to P_j is now said to be possible under m with input Q_k if $j \in J_k(i, m, \mathbb{P})$.

The new transition relation $\delta(m, \mathbb{P}, \mathbb{I})$ refines $\delta(m, \mathbb{P})$ in the sense that

$$J(i, m, \mathbb{P}) = \bigcup_{k \in \mathbb{I}} J_k(i, m, \mathbb{P}),$$

and the new transitions are therefore possibly less non-deterministic than the old ones.

2.2.4 Product Partitions

Finer partitions result in possibly more deterministic transitions. By considering the cross-product of all class-scoped partitions of a class, the algorithm in Section 2.2 can be made to work with the finest partition for the class under analysis.

The *cross-product partition* of a class is the Cartesian product of the state spaces of all class-scoped partitions belonging to the class. The cross-product is itself a class-scoped partition. For instance, in the Calculator example $\mathbb{P}_1$ partitions the value space of the `newNumber` variable while $\mathbb{P}_2$ is a three state partition including the `adding` and `subtracting` fields. The cross-product partition for the `CalcEngine` class is $\mathbb{P}_1 \times \mathbb{P}_2 =$
$\{adding \wedge newNumber,$
$\neg adding \wedge newNumber \wedge subtracting,$
$\neg adding \wedge newNumber \wedge \neg subtracting,$
$adding \wedge \neg newNumber,$
$\neg adding \wedge \neg newNumber \wedge subtracting,$
$\neg adding \wedge \neg newNumber \wedge \neg subtracting\}.$
In this case partitions $\mathbb{P}_1$ and $\mathbb{P}_2$ are disjoint with respect to their variables. In general, when all partitions participating in the product are disjoint from each other with respect to their variables, the transition relation for the product space is just the Cartesian product of the individual transition relations. Therefore, state transitions within the cross-product are completely determined by the component transitions in this case, and taking the cross-product does not refine our insight into the behavior of the class.

A cross-product partition is more interesting for a class whose partitions involve an overlapping set of variables. The product formed over non-overlapping partitions may contain states and transitions that any of its component partitions individually fail to specify. For instance, the states and transitions observed of the cross-product partition in the example presented later in Section 3.2 correspond exactly to the nine `(x, y)` coordinates of that class and the moves between them, even though no single partition was fine enough to do this.

Just as with ordinary partitions, certain states in a product may be unobserved over all the execution traces. In the case of a product of overlapping partitions this can now happen for three possible reasons: either the conjunction specifying the state may be logically false; or, as can happen with other partitions, the state may be inherently unreachable by the class, or it may simply have been avoided by the runtime environment. Although some of the logically false state descriptions can be eliminated by an elementary simplification procedure to be described next, our approach is generally unable to distinguish between these three cases.

In order to eliminate patently false state descriptions and to improve readability of otherwise lengthy formulas by eliminating redundancy where possible, the simplification of state descriptions into their prime implicants [21] has been applied. This simple approach eliminates some logically false formulas and performs well on our examples. For instance, the object invariant for the `Loan` example in Section 3.3 is simplified from six subformulas to just two by this kind of elementary propositional reasoning.

Other than serving as the finest class scoped partition of the class, the product partition may help reveal a heretofore unnoticed refinement relation between two individual partitions of a class. Partition $\mathbb{P}_1$ is a *refinement* of a partition $\mathbb{P}_2$ if and only if each subspace in $\mathbb{P}_1$ refines a subspace in $\mathbb{P}_2$. This is just the familiar notion of one equivalence relation refining another. To establish whether a partition $\mathbb{P}_1$ refines partition $\mathbb{P}_2$, an algorithm would check whether the observed states of the cross-product $\mathbb{P}_1 \times \mathbb{P}_2$ represent a many-to-one mapping of $\mathbb{P}_1$ states to $\mathbb{P}_2$ states. By the nature of dynamic analysis, the conclusion of such a refinement analysis is unsound.

2.2.5 Object Invariants

Complementary to the view that partitions define transition relations on methods is the one that some partitions may also define object invariants. An object invariant is just a formula Φ such that

$$\{\mathit{true}\}m\{\Phi\}$$

is a valid Hoare triple for each method and constructor m.

Each partition $\mathbb{P}$ is constructed so that its states are mutually exclusive. In particular, for each $i, j < |\mathbb{P}|$ the implication

$$P_i \Rightarrow \neg P_j$$

is a tautology whenever $i \neq j$. Although tautologies strictly count as object invariants, tautologous invariants contain no information about the class under analysis. For the purposes of identifying non-tautologous object invariants, implications

$$\mathit{cond}_i \Rightarrow \neg \mathit{cond}_j$$

paralleling the form of those above but arising when P_i and P_j are replaced by their underlying test expressions are considered next.

In order for an implication of this kind to exist $\mathbb{P}$ must be based on an *if-then-else* statement involving two or more user-defined tests. In practice programmers tend to express their tests in highly economical fashion with each test making use of implicit assumptions about the given problem domain and its chosen representation that are not true in all models. Thus, if it holds at all, any pairwise mutual exclusion among the tests is unlikely to be a logically valid truth, but rather one that is heavily implicated by the chosen representation of the class.

Taking the `CalcEngine` class of Section 2.1.1 as an example it will be noted that there is nothing *a priori* valid about the mutual exclusion of *adding* and *subtracting* which are on the surface just two independent boolean variables. Instead it is as a consequence of the significance attributed to each of these variables in the current representation of the `CalEngine` class and the correctness of its overall implementation that this exclusion is found to hold. The current approach considers every pairwise exclusion for a given partition $\mathbb{P}$ in an attempt to discover relevant object invariants for the class under analysis.

In much that same way as the transition relation for $\mathbb{P}$ is approximated over all the runs of a chosen dynamic analysis, a set $\mathcal{E}^*(i, \mathbb{P})$ can be computed for each $cond_i$ approximating the largest set $\mathcal{E}(i, \mathbb{P}) \subseteq \mathcal{I}_{\mathbb{P}} - \{i, |\mathbb{P}|\}$ such that

$$cond_i \Rightarrow \bigwedge_{j \in \mathcal{E}(i,\mathbb{P})} \neg cond_j$$

is an object invariant. Once again this approximation is unsound because the actual maximum $\mathcal{E}(i, \mathbb{P})$ may in fact be smaller. This maximum is, however, well-defined because the empty set makes the above implication a tautology, and whenever two sets satisfy the implication so does their union.

A final way considered under the current approach to obtain an object invariant from a given partition $\mathbb{P}$ is to conjecture that the disjunction

$$\bigvee_{i<|\mathbb{P}|} cond_i$$

of all test expressions is exhaustive in describing the relevant object states for the class. To do so is effectively to conjecture the irrelevance of the *else*-state. The rationale for doing this is that there is always an implicit *else*-state whether the programmer has use for it or not.

It would be possible to go even further and question the relevance of some of the other states for which the programmer actively wrote tests to identify. This possibility is, however, not pursued in the current approach. Instead, it is assumed that every test indicates a relevant object state. Just as with the other object invariants, the status of this conjecture as an actual object invariant can be approximated over a specified dynamic analysis.

2.2.6 Implementation

The state space partitioning approach is implemented in a tool called ContExt which is based on Daikon [1, 24], a general tool for dynamic constraint detection implemented in Java. In ContExt, static analysis extracts test expression groups from the conditional statements belonging to the methods of a target class C. These expression groups are then used to define state partitions for C as described in Section 2.2.1. Each partition participates in the creation of two types of hypotheses: constraints on the transitions relating states at the pre- and postconditions of each method, and object invariants for class C. The partition hypotheses are then suitably encoded to allow them to be dynamically checked in Daikon along with Daikon's universal properties. Thus, ContExt infers both "native" Daikon constraints and the new partition constraints. A post processing step as outlined in Section 2.2.2 lets us infer the transition relation and recover the disjunctive precondition constraints.

Even though ContExt considers all of Daikon's constraints, it may not infer some of the unconditional constraints Daikon alone would have on the same test suite. Due to an increased context sensitivity per program point, which we have implemented by means of an increased number of contextualized program points, a given program point in ContExt may observe fewer data samples than the corresponding program point in Daikon. This may result in some constraints losing their statistical justification at these program points. Since statistical justification of constraints is used to filter out the less likely ones, some of these may fail to be reported by ContExt. Application of a finer test suite will solve this problem.

As is to be expected, the state space partitioning implementation is more expensive in terms of space and time than pure Daikon, but not prohibitively so. In the current implementation the main increase occurs in the number of program points considered for constraint inference. The state space partitioning technique increases the number of program points by about a factor of the size of the largest transition relation. Both time and space complexities of Daikon are linear in the number of program points [24]. Because each program point is more context sensitive than the unconditional one it replaces, ConText typically generates more constraints than Daikon as is discussed in Section 4.1. In practice this added cost has not yet shown itself to be prohibitive, and ConText runs comparably to Daikon.

3. EXTENDED EXAMPLES

This section presents the results of applying the state space partitioning technique to five simple but non-trivial examples. For each example we compare the constraints produced by our approach to the constraints inferred by Daikon on the same test suite. The sensitivity to state context at program points allows our approach to identify an object invariant and construct finite state machines that reflect the state transitions of target objects.

This section views the results of the state space partitioning technique in terms of finite state machines (FSMs). As discussed previously, the technique identifies transitions induced by a method of an object in terms of its states. The state transitions for all methods of an object can be combined into a finite state machine (FSM) which then represents the behavior of this object as a whole. The FSM for objects of class `A` is constructed as follows: the subspaces identified by state partitions of `A` are used as its states and the names of methods in class `A` correspond to the input alphabet of the FSM. The transitions of the FSM are specified by the state space partitioning technique.

3.1 Calculator Example

We start with the Calculator example which originates from an introductory course in object-oriented programming and is described in detail in section 2.1.1. Figure 5 presents the FSM for the `CalcEngine` inferred by the state space partitioning approach. In particular, the FSM highlights that, for example, the `plus` operation puts a `CalcEngine` object into the `adding` state independently of its initial state. The `numberPressed` method serves as an identity function on the $\mathbb{P}_2$ partition, but causes any $\mathbb{P}_1$ state to transition into the $\neg newNumber$ state (P_2). Any other method transitions each $\mathbb{P}_1$ state into P_1.

Even though in this example Daikon infers single variable postconditions which are comparable to the consequents specified by the state partitioning technique, it does not capture the state transitions induced by a method, nor does it identify the implicit state

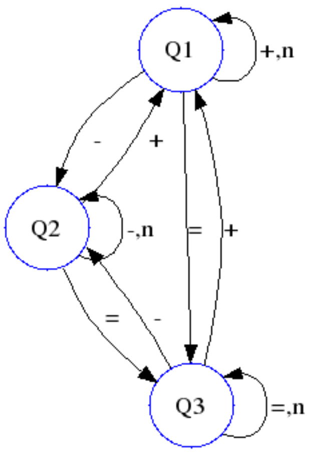

Figure 5: Transition diagram for `CalcEngine` with two state partitions: $I\!P_1 = \{P_1, P_2\} = \{newNumber, \neg newNumber\}$, $I\!P_2 = \{Q_1, Q_2, Q_3\} = \{adding, \neg adding \wedge subtracting, \neg adding \wedge \neg subtracting\}$, where +, −, = and `n` respectively correspond to `plus`, `minus`, `equals` and `numberPressed` methods of the `CalcEngine` class.

$$
\begin{array}{ll}
I\!P_1 = \{P_1, P_2\} & P_1 \equiv x > 0 \\
 & P_2 \equiv x \leq 0 \\
I\!P_2 = \{Q_1, Q_2\} & Q_1 \equiv x < 2 \\
 & Q_2 \equiv x \geq 2 \\
I\!P_3 = \{R_1, R_2\} & R_1 \equiv y > 0 \\
 & R_2 \equiv y \leq 0 \\
I\!P_4 = \{S_1, S_2\} & S_1 \equiv y < 2 \\
 & S_2 \equiv y \geq 2 \\
I\!P_5 = \{T_1, T_2, T_3, T_4\} & T_1 \equiv y = 0 \\
 & T_2 \equiv y = 2 \wedge y \neq 0 \\
 & T_3 \equiv x = 1 \wedge y \neq 2 \wedge y \neq 0 \\
 & T_4 \equiv x \neq 1 \wedge y \neq 2 \wedge y \neq 0
\end{array}
$$

Figure 6: Partitions for the `Puzzle` class

machines which control the behavior of `CalcEngine`. By its nature the state space partitioning technique identifies the parts of the state that control the behavior of an object and therefore reveals the state variables whose value changes are of interest. Because ContExt subsumes Daikon, conditional constraints on state variables also enable it to infer the changes that occur in other variables. An example is the constraint on the `displayValue` field at the post-condition of the `numberPressed` method $orig(\neg newNumber) \Rightarrow displayValue == 10 * orig(displayValue) + orig(number)$.

The state space partitioning approach also reports disjunctions of observed subspace partitions at the preconditions of the methods of the `CalcEngine` class and the essential object invariant $(\neg adding \vee \neg subtracting)$ as discussed in more detail in section 2.1.1. Since a generalized disjunction would be too expensive to compute, Daikon's grammar of properties omits it. This inexpressibility prevents Daikon from inferring these constraints on the `CalcEngine`.

3.2 Puzzle Example

The next example is a `Puzzle` class which represents an environment with an agent. The environment is a 3x3 board which tracks the location of the agent with the `x` and `y` fields and the status of two doors at one end of the room. Each door toggles

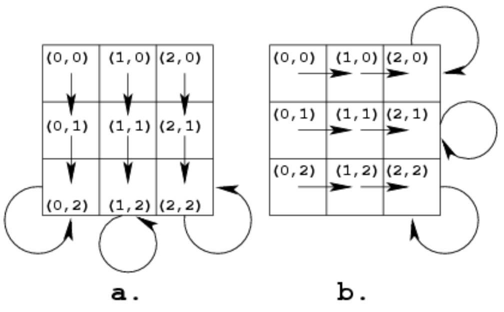

Figure 7: Recovered behavior of the `moveBackward` (a) and `moveRight` (b) methods of the `Puzzle`

```
if (expiry == null) { ... }
else if (maturity == null) { ... }
else { ... }

if (expiry != null && maturity != null) { ... }
else if (maturity != null) { ... }
else if (expiry != null) { ... }
```

Figure 8: Conditional statements in the `Loan` class that identify three different kinds of loans based on the tests on `expiry` and `maturity`

from open to closed depending on its previous state and where the agent moves next. The agent is allowed to move left, right, forward and backward in the environment. The moves respectively correspond to `moveLeft`, `moveRight`, `moveForward`, and `moveBackward` methods of the `Puzzle` class. The goal is for the agent to cross the room to the location of the doors with both doors open. This example was given as a homework assignment in a programming class. Figure 6 depicts the partitions constructed for `Puzzle` by the state space partitioning technique and Figure 7 presents the recovered specification.

None of the partitions in Figure 6 alone enables our approach to characterize the moves of the agent precisely. It is the *cross-product partition* constructed over all state space partitions of the `Puzzle` that results in the precise specification of the "move"-methods of the `Puzzle` as depicted on Figure 7. Each conjunction in the cross-product partition identifies either a single square on the game board, or an impossible state (i.e., false). Also, $I\!P_5$ is seen to refine $I\!P_3$ and $I\!P_4$.

On this example, Daikon is able to infer constraints that approximate the behaviors of the "move"-methods, but do not specify them precisely. For example, Daikon infers that the `x` coordinate of a player does not change and `y one of {1, 2}` at the postcondition to the `moveBackward` method. These constraints, however, do not capture that the `moveBackward` method always moves a player only *one* square back. The comparison with Daikon is discussed further in Section 4.1.

3.3 Bank Loan Example

Next we report on the results the state partitioning approach produced for the Bank Loan example. This example is presented in the "Refactoring to Patterns" book [16] in order to demonstrate the application of the *Replace Conditional Calculations with Strategy* pattern. In this example the author starts with the `Loan` class that has two fields `expiry` and `maturity` of type `Date` which deter-

$I\!\!P_1 = \{P_1, P_2, P_3\}$	$P_1 \equiv expiry = null$
	$P_2 \equiv maturity = null \wedge expiry \neq null$
	$P_3 \equiv maturity \neq null \wedge expiry \neq null$
$I\!\!P_2 = \{Q_1, Q_2, Q_3, Q_4\}$	$Q_1 \equiv expiry \neq null \wedge maturity \neq null$
	$Q_2 \equiv maturity \neq null \wedge expiry = null$
	$Q_3 \equiv expiry \neq null \wedge maturity = null$
	$Q_4 \equiv expiry = null \wedge maturity = null$

Figure 9: Partitions for the `Loan` class

$$
\begin{aligned}
&I\!\!P = \{Q_1, Q_2, Q_3, Q_4, Q_5, Q_6\}\\
&Q_1 \equiv state = INIT\\
&Q_2 \equiv state = SP\_ONE\\
&Q_3 \equiv state = SP\_TWO\\
&Q_4 \equiv state = EP\_ONE\\
&Q_5 \equiv state = EP\_TWO\\
&Q_6 \equiv state \neq (INIT \vee SP\_ONE \vee SP\_TWO \vee\\
&\qquad EP\_ONE \vee EP\_TWO)\\
&I\!\!I = \{T_1, T_2, T_3\}\\
&T_1 \equiv mouse = DOWN \wedge button = LEFT\\
&T_2 \equiv mouse = UP \wedge button = LEFT\\
&T_3 \equiv mouse \neq (UP \vee DOWN) \wedge button \neq LEFT
\end{aligned}
$$

Figure 10: Partitions for the `mouseEvent` method. $I\!\!P$ is a class scoped partition. $I\!\!I$ is an input scoped partition

mine three different kinds of loans: a **term loan**, when `expiry` is `null` and `maturity` is not `null`; a **revolver loan**, when `expiry` is not `null` and `maturity` is `null`; and a **RCTL loan**, when both `expiry` and `maturity` are not `null`. The key observation required of the reader is to recognize that the two conditional statements in Figure 8 identify the same three types of loans, which is not obvious from the way the conditional statements are written. Critical to this is the observation that the code for `Loan` assumes that a `Loan` object can never be in a state when both `expiry` and `maturity` are `null`.

Given the two conditional statements in Figure 8 the state space partitioning approach considers two partitions, $I\!\!P_1$ and $I\!\!P_2$ presented on Figure 9. The simplified cross-product of the two partitions constructed by our approach is $I\!\!P_1 \times I\!\!P_2 = \{\langle P_1 \wedge Q_2\rangle, \langle P_1 \wedge Q_4\rangle, \langle P_2 \wedge Q_3\rangle, \langle P_3 \wedge Q_1\rangle\}$. This cross-product reveals that partition $I\!\!P_2$ is a refinement of partition $I\!\!P_1$, because Q_2 and Q_4 refine P_1. This, combined with the fact that Q_4 is never observed is the main observation that the author of the example expects the reader to make. Furthermore, our approach discovers that a `Loan` object does not make any transitions on the identified partitions. Such a case, when an object does not make any transitions on any of its partitions, allows our tool to hypothesize that the target class is a “union” class, meaning that the target class combines the functionality of several classes in itself. Also, the state space partitioning technique infers the essential object invariant for `Loan`, $expiry \neq null \vee maturity \neq null$, which reflects that a `Loan` object can never have both `expiry` and `maturity` equal to `null`.

In this example again Daikon is limited by the fact that disjunction is not in its language, and it is unable to make any of the important inferences about the relationship of these variables to `null`.

3.4 Mouse Event Handler Example

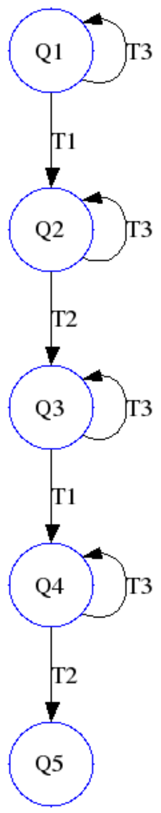

Figure 11: Recovered behavior of the `mouseEvent` method in the `RectangleHandler` class

Our next example originates from a state-based class, `RectangleHandler`, used to handle mouse events for drawing two rectangles consecutively. This example originates from a project assignment for a graphics course.

In this example, the state space partitioning technique considers the class scoped partition $I\!\!P$ and input scoped partition $I\!\!I$ for the `mouseEvent` method that performs that state transitions for the `RectangleHandler` class. It is the refinement provided by the input scoped partition $I\!\!I$ on the class scoped partition $I\!\!P$ that enables our approach to accurately specify the transitions induced by the `mouseEvent` method as presented in Figure 11. Without this refinement, our approach can only infer non-deterministic transitions for the states of $I\!\!P$, such as $Q_1 \rightarrow \{Q_1, Q_2\}$.

3.5 ATM Example

The Automated Teller Machine simulation [2] was written by Dr. Bjork as an example of good object-oriented design and programming practices. The author used the state chart in Figure 12 to specify the transitions between the states for a single user session with an ATM. The states and transitions highlighted in green on Figure 12 refer to the states and transitions recovered by the state space partitioning technique. As the diagram suggests, the technique was able to identify the states and the corresponding transitions specified by the author of the simulation. Since the algorithm is, however, not concerned with identifying which states are initial or terminal in the FSMs it constructs, ConText does not actually label these for the ATM.

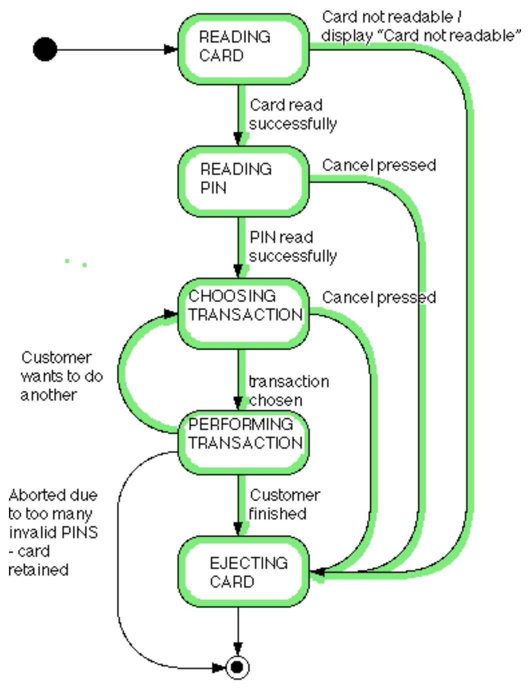

Figure 12: State chart for a single ATM session. Recovered states and transitions are highlighted.

Daikon is not able to infer the transitions for the Mouse Event Handler and the ATM examples. In these two examples a variety of states are observed at both preconditions and postconditions to the method that performs the transitions. Therefore a simple unconditional equality check between the visible variables is insufficient to capture the behavior of this method. Being once again hindered by its inability to observe disjunctive constraints, Daikon does not recognize the ATM and the `RectangleHandler` for the state machines that they are.

4. EVALUATION

The five examples presented in the last section provide a qualitative evaluation for our approach. Due to the subjective nature of a specification in terms of constraints, objective quantitative evaluation of a constraint inference technique presents a challenge. This section briefly introduces a methodology for a quantitative evaluation of constraint inference techniques and then presents a case study of the comparative evaluation of the state space partitioning technique versus Daikon on the `Puzzle` example.

Our evaluation methodology is based on a modeling language called Alloy [14]. We use Alloy to create a model of the class under analysis, constrain the behavior of the model, and then check the essential specification against the model. The model for the class is defined in terms of several Alloy signatures, which can be automatically derived from the class fields and method signatures in the source code. Alloy facts constraining the model are automatically translated from the constraints produced by a particular constraint inference tool (e.g., ContExt or Daikon). Each constraint corresponds to exactly one Alloy fact. In this way two Alloy models are created for the target class, one for each tool. Each model uses the same signatures but is constrained by the automatically inferred constraints produced by its respective tool.

	number of assertions	number of checked assertions	number of facts
Daikon	35	18 (51%)	35
Daikon (w/split)	35	23 (66%)	124
ContExt	35	28 (80%)	554

Figure 13: Comparative evaluation of the inferred constraints for ContExt and Daikon on the `Puzzle` example

In order to evaluate the recovered specification, one needs the "goal" set of constraints that define the essential specification of the class. By its nature, this "goal" set of constraints has to be produced manually. The essential specification can be manually translated into Alloy unit assertions. A unit assertion is a statement about a single aspect of behavior. For example, the fact that the `moveBackward` method in the `Puzzle` class does not affect the `x`-coordinate of the agent is a unit assertion. Alloy assertions provide a high-level abstraction from the constraints expressed in the language of a particular inference tool.

The Alloy solver is then used to check each assertion about the expected behavior of the model against the facts produced by a particular inference tool. If Alloy does not find a counterexample to an assertion, the essential behavior corresponding to this assertion is said to have been *recovered* by the tool. This approach allows us to evaluate quantitatively how much of the essential behavior is captured by a particular constraint inference tool. It provides a means to measure the completeness of the recovered specification, also known as recall. We also plan to extend this procedure in order to be able to measure the correctness, or precision, of the inferred specification by approximating the minimum set of facts that enable Alloy to check the assertions.

The advantages of the presented methodology include abstraction from the languages of particular constraint detection tools, higher-level specification of the essential behavior of the class under analysis, and an objective criterion for the evaluation of the recovered specification.

4.1 Comparative Evaluation of the Puzzle

For a preliminary quantitative comparison of Daikon and ContExt on the `Puzzle` example introduced in Section 3.2, both tools were run with the same test suite. An Alloy model with signatures and assertions for the `Puzzle` example was constructed manually. One copy of this model was constrained with the facts inferred by Daikon to produce Daikon's recovered specification. Another copy of this model was constrained with the facts inferred by ContExt to produce ContExt's recovered specification. Comparison of the two specifications is presented in the first and third rows of Figure 13.

The second row shows the results for Daikon when its native splitting mechanism is enabled. At first we used Daikon's own static analysis tool [7] to generate context sensitive program points. This caused Daikon to use the boolean expressions in the `Puzzle` to establish context at points corresponding to the methods where the expressions were found. Somewhat surprisingly, the outcome was the same as for pure Daikon. Next we distributed the statically established contexts at program points non-local to the expressions in order to mimic class scoped partitions. These results are shown in the second row.

Figure 13 shows that in this example ContExt recovered about 60% more of the essential behavior than pure Daikon at the cost of increasing the number of inferred constraints by a factor of about 16. In between these results is row two which corresponds to the naive mimicking of the state space partitioning technique in Daikon. This row illustrates an improvement of 30% over pure Daikon at the cost of increasing the number of inferred constraints by a factor of 3.5. These results seem to suggest that even a naive application of the state space partitioning technique considerably improves specification recovery.

They also show that attempts to improve recovery often result in an additional cost in terms of the number of constraints reported. Such increases will undoubtedly obscure the recovered specification for a human reader, but they do not seem to obstruct Alloy's reasoning about them. If the results can be viewed through the higher abstraction of such tools, then the cost of the increased number of constraints is counterbalanced by a more complete specification.

5. DISCUSSION OF OUR APPROACH

The state space partitioning technique is primarily a dynamic analysis which results in the unsoundness of the reported constraints. When a transition on a state is missing, the technique is unable to distinguish between truly unreachable states and states omitted by the test suite. The unsoundness of the transitions results in reporting potentially stronger constraints as is detailed in Section 2.2.2.

Even though state partitions are derived from the source code of a class, the state space partitioning technique is *not* limited to static facts. The dynamic nature of the analysis enables it to reveal behavior that is not directly present in the program's text. For example, preconditions reveal the "use cases" of a method, and cross-product partitions reflect the possible combinations of state subspaces for different variables. Neither type of fact is obvious from the source code.

The state space partitioning technique potentially requires finer tests. To infer a transition for a particular state, our approach needs to observe this state at runtime. The state space partitioning technique produces more complete results for a test suite which exercises each state subspace at the precondition to each method[1] of an object. On one hand writing such a test suite may seem like it places a significant burden on the programmer; on the other hand a good test suite should cover all execution paths of a class. From the point of view of test coverage, our technique is biased towards better test suites.

To conclude we would like to note that the state partitioning technique is particularly useful for classes that represent explicit or implicit state machines as well as union classes. The technique is designed to specify state transitions, which allows it to capture the behavior of classes that function as state machines. Union classes are classes that represent several types of objects, where each type has a distinct behavior. Typically such a class will attempt to identify the type of the current object by means of conditional checks on its state so that the class can act appropriately for each particular type of object. Of course, practices of good design suggest the application of "Replace Conditional with Polymorphism" refactoring to a union class. The state space partitioning approach identifies state partitions based on the conditional tests in the source code but also discovers the absence of transitions between the partitions. This allows it to hypothesize that the target class is a union class. In the future versions of ContExt such an hypothesis will result in a suggested application of the "Replace Conditional with Polymorphism" refactoring.

[1]Provided that each state subspace is permitted at the precondition to each method of the class by the underlying logic of this class and its intended environment, which may not always be the case.

6. RELATED WORK

This section presents the work related to state space partitioning that was not discussed earlier in this paper.

Csallner et al. [5] also investigated the combination of static and dynamic analysis methods in order to infer program-specific constraints in a tool called DySy. The DySy authors employ a dynamic symbolic execution technique, which performs symbolic execution over an existing test suite. Preconditions and postconditions are then inferred from the path-conditions and symbolic variable values constructed by a symbolic execution over the program's test suite. The constraints inferred by this approach may overlap with those produced by state space partitioning. It is, however, unlikely that either set will contain the other. Since state space partitioning transplants context that it finds in one part of a class to other parts, it is likely to infer certain pre- and postconditions that an approach like DySy, which holds to the letter of the source code, cannot. On the other hand, symbolic execution is likely to net DySy many program specific expressions that would prove useful for full specification recovery. Further comparison of the results produced by the two tools on the same programs certainly deserves a closer investigation.

With Perracotta, Yang et al. [29, 30] focus on dynamic inference of temporal properties. Temporal properties specify the order of occurrence of program events. Temporal properties can also be thought of as transitions, and as such are similar to the postconditions inferred by our approach. However, similarly to Engler et al. [8], the authors of Perracotta are interested in a small number of error-revealing property templates. The state space partitioning technique, in contrast, pursues the different goal of providing a flexible program-specific language for constraint inference which captures the essential behavior of programs.

The structure of the property spaces used for state space partitioning resemble those favored by pure static analysis [4]. Namely, the set of all disjunctions of subspaces associated with a particular partition forms a finite lattice of predicates that are ordered by logical entailment. The transitions computed by the algorithm in Section 2.2.2 can be viewed as a dynamic approximation of each method's semantics as a predicate transformer [3] upon such a lattice. The most complete account of a purely static approach to object-oriented constraint inference is Logozzo's [18]. As is characteristic, his analysis is sound. The example programs he considers are comparable to our examples in Section 3. Given the nature of the static and dynamic approaches, the constraints inferred by Logozzo are likely to be more approximate in certain cases than ours. A detailed evaluation of the results produced by the two techniques on the same examples would be interesting to conduct.

A lot of recent work has been done in the field of extracting models or specifications using either pure-dynamic techniques or a combination of dynamic and static methods. These approaches are quite different from the state space partitioning technique either in terms of their goals, implementation strategies, or results.

Whaley et al. applied dynamic and static methods to infer finite state machine models for correct method call sequences to ensure the correct usage of an API [27]. Henkel and Diwan developed a tool that discovers high-level algebraic specifications from Java classes in the form of axioms using dynamic analysis [12, 13]. Caffeine [10], developed by Guéhéneuc et al., is a tool for dynamic analysis of Java programs, which allows the developer to check conjectures about the behavior of a Java program.

Other research concentrates on aiding in software error detection. Hangal and Lam created DIDUCE - a tool which dynamically formulates constraints for a program and can inform the user when the formulated constraints are violated at runtime [11]. Pytlik et al. [25] built an automatic debugging tool called Carrot based on Daikon. Liblit et al. have developed a general sampling infrastructure for gathering information from executions from multiple users [17] in order to use the data traces to discover software bugs.

7. CONCLUSIONS

Fully specifying the essential behavior of an arbitrary program remains beyond the state-of-the-art automatic techniques. On the bright side, our work demonstrates that a combination of static and dynamic methods can be used to make a step towards this goal. The state space partitioning technique described in this paper devises a flexible language for constraint inference which provides for detection of program-specific disjunctive properties. Our examples show that the technique performs well on classes that function as explicit or implicit state machines as well as union classes.

Our future work involves a more extensive qualitative and quantitative evaluation of the state space partitioning technique. We also plan to investigate other source code fragments that can be mapped to other assumptions about the program.

8. REFERENCES

[1] Daikon invariant detector. `http://pag.csail.mit.edu/daikon`.

[2] R. Bjork. Automated teller machine simulation. `http://www.math-cs.gordon.edu/courses/cs211/ATMExample/`.

[3] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77*, pages 238–252, 1977.

[4] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL '79*, pages 269–282, 1979.

[5] C. Csallner, N. Tillmann, and Y. Smaragdakis. Dysy: Dynamic symbolic execution for invariant inference. Technical Report MSR-TR-2007-151, Microsoft Research, November, 2007.

[6] E. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, N.J., 1976.

[7] N. Dodoo, L. Lin, and M. Ernst. Selecting, refining, and evaluating predicates for program analysis. Technical Report MIT-LCS-TR-914, MIT Laboratory for Computer Science, Cambridge, MA, July 21, 2003.

[8] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, 2001.

[9] 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 Programming*, 2007.

[10] Y. Guéhéneuc, R. Douence, and N. Jussien. No Java without caffeine – a tool for dynamic analysis of Java programs. In W. Emmerich and D. Wile, editors, 17^{th} *conference on Automated Software Engineering*, pages 117–126. IEEE Computer Society Press, September 2002.

[11] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, New York, NY, USA, 2002. ACM Press.

[12] J. Henkel and A. Diwan. Discovering algebraic specifications from java classes. In L. Cardelli, editor, *ECOOP 2003 - Object-Oriented Programming, 17th European Conference*, Darmstadt, July 2003. Springer.

[13] J. Henkel and A. Diwan. A tool for writing and debugging algebraic specifications. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 449–458, Washington, DC, USA, 2004. IEEE Computer Society.

[14] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[15] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin. Automated support for program refactoring using invariants. In *ICSM 2001, Proceedings of the International Conference on Software Maintenance*, pages 736–743, 2001.

[16] J. Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 2004.

[17] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, San Diego, California, June 9–11 2003.

[18] F. Logozzo. Class invariants as abstract interpretations of trace sematics. *Computer Languages, Systems and Structure*, in press, 2006.

[19] T. Ne Win. Theorem-proving distributed algorithms with dynamic analysis. Master's thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 2003.

[20] T. Ne Win, M. D. Ernst, S. J. Garland, D. Kırlı, and N. Lynch. Using simulated execution in verifying distributed algorithms. *Software Tools for Technology Transfer*, 6(1):67–76, July 2004.

[21] R. J. Nelson. Simplest normal truth functions. *The Journal of Symbolic Logic*, 20(2):105–108, 1955.

[22] J. Nimmer and M. Ernst. Invariant inference for static checking: An empirical evaluation. In *Proceedings of the ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE 2002)*, pages 11–20, Charleston, SC, November 20–22, 2002.

[23] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pages 232–242, Rome, Italy, July 22–24, 2002.

[24] J. H. Perkins and M. 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, Newport Beach, CA, USA, November 2–4, 2004.

[25] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss. Automated fault localization using potential invariants. In *AADEBUG'2003, Fifth International Workshop on Automated and Algorithmic Debugging*, pages 273–276, Ghent, Belgium, September 8–10, 2003.

[26] O. Raz, P. Koopman, and M. Shaw. Semantic anomaly detection in online data sources. In *ICSE'02, Proceedings of the 24th International Conference on Software Engineering*, pages 302–312, Orlando, Florida, May 22–24, 2002.

[27] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. *SIGSOFT Softw. Eng. Notes*, 27(4):218–228, 2002.

[28] T. Xie and D. Notkin. Tool-assisted unit test generation and selection based on operational abstractions. *Automated Software Engineering Journal*, 2006.

[29] J. Yang. Automatically inferring temporal properties. In *ICSE '05: The Doctoral Symposium, 27th International Conference on Software Engineering*, St. Louis, Missouri, USA, May 15–21, 2005.

[30] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining temporal api rules from imperfect traces. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, Shanghai, China, May 20–28, 2006.