Extending Dynamic Constraint Detection with Disjunctive Constraints

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

The languages of current dynamic constraint detection techniques are often specified by fixed grammars of universal properties. These properties may not be sufficient to express more subtle facts that describe the essential behavior of a given program. In an effort to make the dynamically recovered specification more expressive and program-specific we propose the state space partitioning technique as a solution which effectively adds program-specific disjunctive properties to the language of dynamic constraint detection. In this paper we present ContExt, a prototype implementation of the state space partitioning technique which relies on Daikon for dynamic constraint inference tasks.

In order to evaluate recovered specifications produced by ContExt, we develop a methodology which allows us to measure quantitatively how well a particular recovered specification approximates the essential specification of a program's behavior. The proposed methodology is then used to comparatively evaluate the specifications recovered by ContExt and Daikon on two examples.

Categories and Subject Descriptors

F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Invariants, Specification techniques*

General Terms

Algorithms

Keywords

dynamic constraint inference, disjunctive constraint, behavioral specification

1. INTRODUCTION

Dynamic constraint detection is a dynamic program analysis which strives to recover (part of) a program's specification in the form of constraints. The languages of current dynamic constraint detection techniques are often specified by fixed grammars of universal properties [7, 13]. While a fixed universal language serves well for problems which require the discovery of a well-defined set of problem-specific, but program-independent properties, it may be insufficient to capture the logic of a particular program.

```
if (x < 0) {...}                 P1 ≡ x < 0,
else if (y > 0) {...}    ⇒       P2 ≡ x ≥ 0 ∧ y > 0,
else {...}                       P3 ≡ x ≥ 0 ∧ y ≤ 0
```

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

Our goal is to extend dynamically recovered specifications with program-specific properties that capture the subtle essential properties of a program under analysis automatically. With this goal in mind, we propose the state space partitioning technique which combines static and dynamic program analysis to *automatically specialize* the language of constraint detection to a particular program on a per-program basis. The key observation for the technique is that certain constructs from the source code can be mapped to the assumptions about the target program. Such assumptions are then used to infer likely constraints on partitions by observing execution traces.

The constraints produced by the technique are 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. 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[1] 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 $x \geq 0 \wedge y \leq 0$. Once these spaces are identified for a class, then methods of this class can be viewed as potentially inducing transitions between state subspaces.

In such a way the state space partitioning technique introduces a number of different types of *disjunctive constraints* into the language of constraint detection. Disjunctive constraints based on state space partitions include an object invariant, constraints on distinct behavior for each abstract state, as well as constraints on transitions between abstract states induced by the methods of a class. Our object invariant is of the form $\neg a \vee \neg b$ which says that properties a and b are mutually exclusive. Transitions induced by a

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

WODA – Workshop on Dynamic Analysis, July 21, 2008

[1]Assuming 32 bit integer representation.

method m are depicted as $p \Rightarrow q$, where p is an abstract state on variables at precondition of m while q is a disjunction of abstract states on variables at postcondition of m. Our preliminary evaluation results in Section 4 seem to suggest that such disjunctive constraints provide for models that capture the essential behavior of certain programs more completely than traditional dynamic analysis as implemented in Daikon which avoids general disjunctive relations since they are expensive to compute.

In this paper we provide an overview of the state space partitioning technique and describe its prototype implementation 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.

The rest of this paper is organized as follows. The next section presents a motivational example, introduces the state space partitioning technique, details the implementation of the technique in ContExt and concludes with comparative complexity analysis. Section 3 discusses the limitations of our approach. Section 4 introduces an evaluation methodology and then applied it to two examples in order to comparatively evaluate the recovered specifications produced by ContExt and Daikon. Section 5 presents related work. And Section 6 concludes.

2. OUR APPROACH

We start this section with a simple, but illustrative example for the state space partitioning technique which serves as both our motivational example and running example for the rest of the section.

2.1 The Calculator Example

The `CalcEngine` class in Figure 2a 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 2b. 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.

Postconditions 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 postcondition 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.

```
public class CalcEngine {

//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;

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;
  ...
}

public void clear() { ... }

public void plus() { ... }

public void minus() { ... }

}
```

a. Code fragment for the `Calculator` class

Object Invariant:

```
context CalcEngine inv:
(!this.adding || !this.subtracting)
```

Method Constraints:

```
context CalcEngine::numberPressed(int number)
pre: P1 || P2, Q1 || Q2 || Q3
post: 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))
context CalcEngine::clear()
pre: P1 || P2, Q3
post: orig(P1) ==> P1, orig(P2) ==> P1,
 orig(Q3) ==> Q3
```

b. Constraints inferred by ContExt

Figure 2: Calculator Example

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.

Next we proceed to present the overview of the state space partitioning technique. The `Calculator` example is used as a running example to illustrate the key concepts.

2.2 Overview of the State Space Partitioning Technique

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. The technique starts by forming disjoint partitions of the state spaces of the program variables involved in expressing these tests. Such a partition is obtained by conjoining each test with the negations of all the tests preceding it in the sequence. To account for the complete state space of the involved variables, we form an explicit `else`-partition that is the combined negation of all the tests in the `if-then-else` sequence. Thus, an `if-then-else` statement with n tests results in $n+1$ disjoint state partitions. For example, `adding` followed by `subtracting` defines the sequence of boolean expressions from the `if-then-else` statement of the `equals` method of the `CalcEngine` class. Following the above procedure, the tests are conjoined into three disjoint partitions $Q_1 = \texttt{adding}$, $Q_2 = \neg\texttt{adding} \wedge \texttt{subtracting}$, and $Q_3 = \neg\texttt{adding} \wedge \neg\texttt{subtracting}$. Let us also note that a partition, such as Q_1, is used to denote both a logical formula, $adding\ is\ true$, and the respective subspace of all object states where the `adding` attribute is true.

The state space partitioning technique considers partitions of two scopes: class and input. *Class-scoped* partitions are expressed in terms of instance variables and constants alone and can be evaluated anywhere in the class. *Input-scoped* partitions also contain a variable which serves as an input parameter to the method from which the partition has been extracted, and as such, they can only be evaluated in the context of this method. In the `Calculator` example all partitions are expressed in terms of `CalcEngine`'s attributes and are therefore class-scoped.

Intuitively, the partitions constructed above represent the abstract states explicitly identified by the developer with the conditional logic of the class. Having identified potentially interesting partitions in terms of program variables, we now define the preconditions, postconditions and object invariants based on these partitions. The preconditions are designed to identify the use cases of a particular method. A precondition is then represented by a disjunction of states, $P_1, ..., P_n$, from the same partition space, $\bigvee_{j \in [1..n]} P_j$. For instance, $Q_1 \vee Q_2 \vee Q_3$ is the precondition to the `plus` method of the `CalcEngine` class.

The postconditions considered by the state space partitioning approach take the form of transitions on the identified abstract states induced by a particular method of a class. For example, the `plus` method induces the transition from the abstract state Q_2 into the abstract state Q_1 in the `CalcEngine` object, denoted as $Q_2 \Rightarrow Q_1$.

Object invariant expressions are designed to check whether the tests of the corresponding `if-then-else` statement are mutually exclusive. For this purpose object invariants are constructed on the test expressions from an `if-then-else` statement rather than partitions. A hypothesized object invariant is a disjunct that asserts that each test in the sequence of boolean tests is disjoint from the others. For example, the object invariant hypothesis for the space $\mathbb{P}_2$ checks for the mutual exclusion of `adding` and `subtracting` attributes and is denoted by $(adding \wedge \neg subtracting) \vee (\neg adding \wedge subtracting) \vee (\neg adding \wedge \neg subtracting)$.

The constraints presented above are all disjunctions. If a disjunctive template were to be used in the computation of disjunctive pre- and postconditions, it would require a number of state combinations exponential in the size of each abstract state space. This approach is computationally prohibitive, instead we designed an algorithm that only considers a linear number of such combinations.

The following algorithm allows a dynamic constraint detector to approximate transitional postconditions with potentially *stronger* ones. Let $\mathbb{P}$ be an abstract space that consists of three abstract states, P_1, P_2, and P_3. To support the inference of state transitions, the initial state is considered over variable values at precondition, denoted as P_i^{pre}, while the disjunctive result is inferred over variable values at postcondition, denoted as P_i^{post} for some $i \in [1..3]$. At the postcondition program point for a method M compute the transitional postcondition for each P_i^{pre}, $i \in [1..3]$, as follows:

1. Assume that $P_i^{pre} \Rightarrow \neg P_1^{post}$, $P_i^{pre} \Rightarrow \neg P_2^{post}$, and $P_i^{pre} \Rightarrow \neg P_3^{post}$ are all possible transitions. Denote this by the set S of indices $S = \{1,\ 2,\ 3\}$.
2. Perform dynamic analysis, and whenever P_i^{pre} and P_j^{post} both hold, remove j from S.
3. Approximate the transitional postcondition for P_i with a disjunction of abstract states whose indices are contained in the complement of S, $P_i^{pre} \Rightarrow \bigvee_{k \in S^c} P_k^{post}$.

The algorithm for precondition inference is analogous to the one above, except that no transitional relation is computed.

The following example is used to provide the intuition behind the algorithm. Suppose that $i = 1$ and after step 2, $S = \{1, 3\}$. This means that $P_1^{pre} \Rightarrow \neg P_1^{post}$ and $P_1^{pre} \Rightarrow \neg P_3^{post}$ are consistent with the observed data. Also, $P_1^{post} \vee P_2^{post} \vee P_3^{post}$ is true by construction. Then, the transition $P_1^{pre} \Rightarrow P_2^{post}$, which is computed by the algorithm, follows by propositional logic.

Currently our approach considers a Cartesian product of the partitions which originate from separate conditionals in order to refine the state space and improve the precision of the inferred constraints. This section presents an overview of the state space partitioning technique. To keep the discussion short, some details, such as the use of product partitions and refined transition relations for postconditions, have been omitted [10].

The main advantage of the technique is the efficient introduction of the program-specific disjunctive constraints into the language of dynamic constraint inference.

2.3 ContExt: Implementation

The implementation for the state space partitioning technique uses lightweight static analysis of Java source code for abstract state extraction. Dynamic analysis tasks are delegated to Daikon [1, 12] which is a general and publicly available tool for dynamic constraint detection implemented in Java. At the end, ContExt combines the constraints inferred by our approach with those inferred by Daikon in its output. For better understanding of our approach as well its advantages and limitations, we start with a short introduction to the dynamic constraint inference mechanism of Daikon. We then proceed on to the description on the implementation for the state space partitioning technique.

The constraints inferred by Daikon are determined by *program points*, a fixed *grammar of properties*, and the variables visible at the various program points. A *program point* is a location within the target program where constraints are inferred. A fixed *grammar of properties* is a list of templates that describe possible relationships between variables. Each template is instantiated with all possible combinations of the program variables of correct type visible at a program point. In the end Daikon reports the *likely* constraints as the instantiated templates that are never invalidated by any data trace.

Daikon attempts to minimize the number of reported constraints. First, it uses statistical justification to distinguish chance relationships from likely constraints. Daikon establishes the properties that hold on the given data trace, and then for each property, it computes the probability that the observed property could have happened by

chance alone on a random set of samples. Only properties whose probability is smaller than the user-defined confidence parameter qualify as likely constraints and are reported. Second, Daikon suppresses constraints that are easily derived from one that is reported.

Daikon outputs two kinds of *likely* constraints: accidental properties and true constraints. The latter are constraints in the traditional sense, whereas accidental properties are an artifact of the values observed during the examined executions and are not universally true for all program runs.

A fixed grammar of properties is used to describe the language of constraints. In general, the grammar is insensitive to the context of a particular program. However, Daikon also supports a mechanism to infer conditional constraints in the form of implications $p \Rightarrow q$ (if p holds then q holds). An antecedent in a conditional constraint can be thought to provide context sensitivity with respect to some state of the target program.

Checking conditional hypotheses for all properties in the grammar of properties is computationally prohibitive. Instead, Daikon uses *splitting conditions*, or *splitters* for short, for the computation of implications. A splitting condition is a boolean expression in terms of some program variables visible at a program point T. A splitter a partitions the data trace into two mutually exclusive subsets: the first subset T_a contains the data values that satisfy the splitting condition a, and the second, $T_{\neg a}$, contains the data values such that the splitting condition a does not hold. After splitting the data, constraints are independently inferred on each set, T_a and $T_{\neg a}$. Then, constraints inferred over T_a and $T_{\neg a}$ are combined into implications based on the key observation that each mutually exclusive constraint implies the other constraints inferred over its own subset of data. The algorithm for the creation of implications in Daikon is described in [5].

In our approach, static analysis extracts the sequence of boolean test expressions $cond_1, cond_2, ..., cond_n$ from each `if`-statement with class-scoped test conditions in the source code for a class C. This sequence is then used to construct a state space partition $I\!P = \{P_1, P_2, ..., P_n, P_{n+1}\}$ for the objects of class C, such that $P_1 = cond_1$, $P_2 = cond_2 \wedge \neg cond_1$, ..., $P_n = cond_n \wedge \neg cond_{n-1} \wedge ... \wedge \neg cond_1$, and $P_{n+1} = \neg cond_n \wedge \neg cond_{n-1} \wedge ... \wedge \neg cond_1$ as described in section 2.2.

After the partitions for class C have been identified, partition-based constraints are created. Each sequence of condition tests drives the construction of an object invariant hypothesis. Each partition participates in the creation of two types of constraints: a disjunctive constraint on the partition states as a precondition to each method of class C and transition constraints between the state partitions as postconditions to methods of class C.

An object invariant initially assumes that all conditions in a sequence are mutually exclusive. As data samples are observed, an object invariant may be weakened so that only some of the conditions are mutually exclusive. In its weakest form, an object invariant says that some conditions always hold throughout the lifetime of the object. For the partition $I\!P$, the strongest form of the object invariant is $(cond_1 \wedge \neg cond_2 \wedge ... \wedge \neg cond_n) \vee (\neg cond_1 \wedge cond_2 \wedge ... \wedge \neg cond_n) \vee (\neg cond_1 \wedge \neg cond_2 \wedge ... \wedge cond_n)$ and the weakest form is $cond_1 \vee cond_2 \vee ... \vee cond_n$.

The inference of transition constraints involves the splitting mechanism of Daikon. Each abstract state P_i from partition $I\!P$ is used as a splitter on the data trace at postcondition program points of the enclosing class. This arrangement provides for convenient checks when P_i^{pre} and P_j^{post} both hold. The algorithm from Section 2.2 is then executed for each splitter P_i from partition $I\!P$ in order to obtain transition relations for P_i. For example let P_1 be a splitting state from the set $I\!P$. Then the data trace for the postcondition of some method M of class C is split into the data samples that satisfy P_1^{pre} and the data samples that satisfy $\neg P_1^{pre}$. Suppose, that method M induces the transformation of P_1 into P_2 or P_3 and both cases are present in the data trace. Then, the disjunctive constraint inferred at the postcondition to method M over the data samples in P_1^{pre} is $P_2^{post} \vee P_3^{post}$. And the transition $P_1^{pre} \Rightarrow P_2^{post} \vee P_3^{post}$ is inferred for the postcondition of method M. Let us also note that Daikon's built-in implication inference is also performed for all splitters that originate from the state space partitions. Daikon-inferred implications potentially provide insight into the state of a class not covered by partitions, which may be referred to as the "what is being controlled" part of the state. For example, the `displayValue` in the `CalcEngine` class serves as the controlled part of the `CalcEngine`'s state.

A precondition to method M is determined to be the disjunction of all abstract states from which a transition was observed.

Since partition creation may result in lengthy formulas that are hard to read, we simplify partition formulas into their prime implicants [11]. This simple approach eliminates some logically false formulas and performs well on our examples. For instance, the object invariant for the `CalcEngine` class is simplified from $(adding \wedge \neg subtracting) \vee (\neg adding \wedge subtracting) \vee (\neg adding \wedge \neg subtracting)$ to $\neg adding \vee \neg subtracting$ with this propositional reasoning.

At the present moment our implementation is capable of evaluating any Java expression on the variables visible to Daikon. Conditional expressions which contain local variables or method calls with parameters are currently ignored. Our future work involves augmenting the implementation to consider conditional tests which contain pure method calls or local variables that can be expressed in terms of class attributes and input parameters. The types of conditional expressions considered by ContExt include `if-then-else` statements as well as `switch`-statements. The need to be able to evaluate arbitrary Java expressions on the data trace prohibits us from using the template mechanism for constraint creation. Instead, a test expression is compiled into Java bytecode and evaluated by the JVM on the supplied variable values.

2.4 Comparative Complexity Analysis

The state space partitioning technique effectively computes disjunctions. This section compares the time and space costs of our approach to those of Daikon.

First let us consider the costs of introducing a generalized disjunctive template into Daikon's grammar of properties. In this case Daikon would have to consider the powerset of all hypothesized constraints at each program point. The total number of hypothesized constraints would then increase to 2^k for each program point, where k is the number of hypothesized non-disjunctive constraints. Obviously, this number of hypothesized constraints is computationally prohibitive.

Second let us compare the time and space complexity of Daikon's algorithm to those of the state space partitioning technique. Let us approximate[2] the space complexity of Daikon's constraint inference with $S = O(P * C)$ and the time complexity with $T = O(P * C * L)$, where P is the number of program points in the target program, C is the number of hypothesized constraints at a program point, and L is the number of data samples observed [12].

The approximate complexities of the state space partitioning approach are presented next. Suppose, there are m class-scoped partitions. Let n be the maximum number of states per class-scoped

[2]For simplicity, we approximate Daikon's space and time complexities with those of the simple incremental algorithm for constraint detection.

partition. Then, the total number of states considered by the state space partitioning technique is $m * n$. So, the state space partitioning approach increases the number of program points, P', at which the constraints are inferred by a factor of $m * n$. The number of hypothesized constraints per program point, C', is increased by an additive factor $m * n$. Therefore, for the state space partitioning technique the relative worst-case space complexity is $S' = P'C' = O(mnP * (mn + C))$ and the relative worst-case time complexity is $T' = O(P' * C' * L) = mn * P * (mn + C) * L$. As expected, the state space partitioning technique is more expensive than plain Daikon inference, but not prohibitively so.

If the technique considers the Cartesian product of all m partitions, then the analysis contends with one partition of size n^m rather than m partitions with n states each. This exponential cost is to be expected for the more path sensitive analysis. Cartesian product analysis is an option that can be used to increase the precision of the inferred constraints at an extra cost.

3. LIMITATIONS

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.

Daikon's implication inference is performed on the splitters which originate from the state space partitioning technique. This inference often provides the insight into the controlled part of the state of a class, however it also results in the increase in the number of accidental constraints reported and loss of precision of the results.

Given the same test suite, the state space partitioning approach may not infer some unconditional constraints that Daikon would. State space partitioning technique infers constraints at split program points. A split program point observes less data samples than the corresponding program point which may result in some constraints being statistically unjustified at a split program point. If a constraint is not statistically justified at a split program point, it will not be reported.

At the moment, we applied the technique only to one class at a time. The future work involves extending the state space partitioning technique to consider the state interactions between composed classes.

4. COMPARATIVE EVALUATION

A quantitative evaluation of constraint inference techniques presents a challenge due to the subjective nature of recovered constraints. In this section we propose a methodology for a quantitative evaluation of constraint inference techniques and then present the results of applying this methodology to comparatively evaluate Daikon and ContExt on several examples.

Our evaluation methodology is based on the Alloy modeling language [9]. First, the class under analysis is modeled in Alloy. Second, the behavior of this model is constrained with inferred constraints, and then the essential specification is checked against the model as depicted in Figure 3. The percentage of the essential specification that Alloy finds valid serves as a quantitative measure of how well a particular recovered specification captures the essential behavior of the class under analysis.

The Alloy model for the class consists of three parts: representation of the class, the essential specification of its behavior, and constraints on the behavior of the model. The representation for the class under analysis 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 inferred 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. To check that the resulting model is not over-constrained, we use a `show` predicate to produce a number of representative non-trivial instances.

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 pressing a number of the `Calculator` after the plus sign results in this number being displayed. Alloy assertions provide a high-level abstraction from the constraints expressed in the language of a particular inference tool.

This step introduces the sensitivity of the evaluation methodology to the subject who performs the translation of the specification into Alloy. The subject may potentially introduce errors or skew the specification in favor of one tool. One possible solution to this problem is to have a neutral person perform the translation. However, in the situation of the absence of such a person with the knowledge of Alloy's language, we to operate under the assumption of a fair translation of the specification.

After the Alloy model has been created, the Alloy solver is 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. In short, we check how many of specification assertions hold given the inferred constraints (facts). 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 Evaluation Case Studies

The two case studies present preliminary results of comparative evaluation of the specifications recovered by ContExt and Daikon based on the methodology introduced in Section 4.

The first case study was performed on the `Puzzle` example, which was given as a homework assignment in a programming class. The `Puzzle` class 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 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

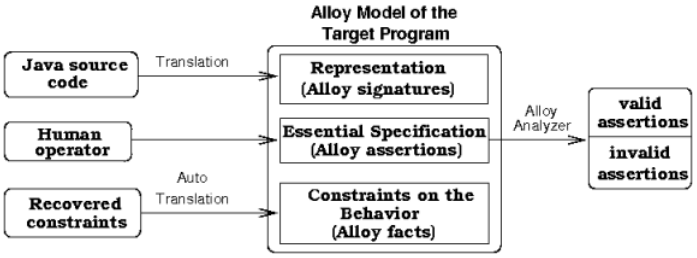

Figure 3: Evaluation technique

Assertion in Alloy	English-language specification

`assert moveForward_1 {`
`all p': Puzzle, p : Puzzle |`
`(p in (p'.moveForward.Unit)) =>`
`(p'.yPosition = p.yPosition <=> p'.yPosition = 0)`
`}`
	The y-coordinate of the agent is to remain the same only when it attempts a moveForward from the top edge of the board (y is 0).

`assert moveForward_2 {`
`all p': Puzzle, p : Puzzle |`
`(p in (p'.moveForward.Unit)) =>`
`(p'.yPosition - 1 = p.yPosition <=> p'.yPosition > 0)`
`}`
	Otherwise, an agent moves forward exactly one square (y-coordinate decreases by one).

`assert moveForward_3 {`
`all p': Puzzle, p : Puzzle |`
`(p in (p'.moveForward.Unit)) =>`
`p.yPosition =< p'.yPosition`
`}`
	The y-position of the agent at the post-condition of the moveForward method is less than or equal to the y-position at precondition.

`assert moveForward_4 {`
`all p': Puzzle, p : Puzzle |`
`(p in (p'.moveForward.Unit)) =>`
`(p.xPosition = p'.xPosition)`
`}`
	Moving forward does not affect the x-coordinate of the agent.

Figure 4: Specification of the `moveForward` method of the `Puzzle`.

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

	number of assertions	number of checked assertions	number of facts
Daikon	15	12 (80%)	55
ContExt	15	15 (100%)	89

Figure 6: Comparative evaluation of the inferred constraints for ContExt and Daikon on the `Employee` example

the `Puzzle` class. The goal is for the agent to cross the room to the location of the doors with both doors open.

Figure 4 presents the specification in terms of the unit assertions in Alloy for the `moveForward` method of the `Puzzle` and their respective translation into the English language. The unit assertions for the `moveLeft`, `moveRight`, and `moveBackward` methods of the `Puzzle` are similar to the `moveForward` method. We believe that the presented assertions provide a fair specification for the `Puzzle` example. These assertions are used on Alloy models for both Daikon and ContExt.

For a preliminary quantitative comparison of Daikon and ContExt on the `Puzzle` example, 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 5.

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 [5] 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 5 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. For instance, while Daikon-recovered facts failed to demonstrate the stronger assertion that an agent moves exactly one square forward by using the `moveForward` method when not on the top edge of the board (assertion `moveForward_2` on Figure 4), they were sufficient to validate the weaker assertion that an agent either moves one square forward or remains on its current square when the `moveForward` is called (assertion `moveForward_3` on Figure 4). The condi-

tional logic of the `moveForward` method distinguishes the agent's locations in which either no move occurs (top row) from the ones where a one square move forward occurs (second and third row). By considering the partitions which are based on the conditional tests, ContExt is able to recover the facts which enable the Alloy Analyzer to show both assertions `moveForward_1` and `moveForward_2`.

Between the results for pure Daikon and ContExt, on Figure 5 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. These results seem to suggest that even a naive application of the state space partitioning technique considerably improves specification recovery.

The second case study is based on the initial design of the `Employee` example from Fowler's Refactoring book [8]. The `Employee` class provides a `payAmount` method which computes the monthly earnings of an employee based on his or her occupation. The same approach was taken for evaluating the recovered specifications for Daikon and ContExt on the `Employee` example as on the `Puzzle` example. The results are presented on Figure 6. On this example, the partitions enable ContExt to distinguish between `Employees` with different occupations and the equations used to compute their respective salaries.

In summary, the results suggest that the inference of disjunctive constraints improves the completeness of the recovered behavioral specification over plain dynamic constraint inference.

5. RELATED WORK

Csallner et al. [3] employ a dynamic symbolic execution technique, which performs symbolic execution over an existing test suite, in order to obtain program-specific constraints. 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.

Engler et al. [6] and Yang et al. [13] focus on a relatively small number of error-revealing properties. The latter consider temporal properties, which can also be thought of as transitions, and as such are similar to the postconditions inferred by our approach. 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.

Dallmeier et al. [4] use a combination of static and dynamic analysis to construct state machines that represent an object's behavior in terms of the object's inspector and mutator methods. Concrete values are abstracted via a state abstraction function. Our approach is similar to that of Dallmeier at al. in that both seek to recover an abstract state model of a program. The difference lies in the state construction techniques and dynamic inference mechanisms. Our approach relies on the conditional logic identified by the developer of a class to provide the states of the automaton, while Dallmeier at al. represent the state with the abstracted return values of inspector methods.

Similar to our work, Arumuga Nainar et al. [2] are interested in finding relevant boolean formulae. In their case, the formulae partition the program state space into only two subspaces, one in which a bug is expressed and the other one in which it is not.

6. 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. The preliminary results suggest that the technique performs well on classes that function as explicit or implicit state machines as well as union classes.

7. REFERENCES

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

[2] P. Arumuga Nainar, T. Chen, J. Rosin, and B. Liblit. Statistical debugging using compound boolean predicates. In *International Symposium on Software Testing and Analysis*, London, United Kingdom, July 9–12 2007.

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

[4] V. Dallmaier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. In *WODA '06: Workshop on Dynamic Analysis*, Shanghai, China, 2006.

[5] 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.

[6] 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.

[7] 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.

[8] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

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

[10] N. Kuzmina, J. Paul, R. Gamboa, and J. Caldwell. State space discovery by guided dynamic analysis. Technical report, University of Wyoming Department of Computer Science technical report UWCS-08-01 (Laramie, WY), March 2008.

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

[12] 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.

[13] 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.