Adapting Side Effects Analysis for Modular Program Model Checking

Oksana Tkachuk
Department of CIS
Kansas State University, USA
oksana@cis.ksu.edu

Matthew B. Dwyer
Department of CIS
Kansas State University, USA
dwyer@cis.ksu.edu

ABSTRACT

There is a widely held belief that whole program analysis is intractable for large complex software systems, and there can be little doubt that this is true for program analyses based on model checking. Model checking selected program components that comprise a cohesive unit, however, can be an effective way of uncovering subtle coding errors, especially for components of multi-threaded programs. In this setting, one of the chief problems is how to safely approximate the behavior of the rest of the application as it relates to the unit being analyzed.

Non-unit application components are collectively referred to as the environment. In this paper, we describe how *points-to* and *side-effects* analyses can be adapted to support generation of summaries of environment behavior that can be reified into Java code using special modeling primitives. The resulting abstract models of the environment can be combined with the code of the unit and then model checked against unit properties. We present our analysis framework, illustrate its flexibility in generating several types of models, and present experience that provides evidence of the scalability of the approach.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model Checking*

General Terms

Verification

Keywords

Modular Flow Analysis, Model Checking, Assume-Guarantee

1. INTRODUCTION

Model checking programs is a rapidly growing sub-field of program analysis (e.g., [4, 8, 18]). Experience analyzing programs with model checking techniques has proven to be useful in finding defects in portions of real code bases. These successes have been enabled in large part by the use of techniques for automating the abstraction of program data (e.g., [8]). Another key factor in these successes is the restriction of analysis to selected parts of a program. One well-studied method for modular model checking is the *assume-guarantee* paradigm [9, 12] where a system is decomposed into two subsystems: one that is modeled explicitly and one that is represented by user supplied specifications that capture assumptions about its behavior. A model checker can then prove properties of the first subsystem under the assumption that the second satisfies its specification. The final step in this paradigm is checking that a subsystem implementation satisfies the stated assumptions.

Most work on assume-guarantee model checking has focused on the patterns of control actions (e.g., process synchronizations) that subsystems expose to one another. Our work adapts the assume-guarantee paradigm to treat the data interaction between a cohesive group of components, called a *unit*, and the rest of the program components, which are collectively termed the *environment*. Whereas traditional assume-guarantee reasoning requires the user to *manually specify* environment assumptions, our approach uses static analysis to *automatically extract* abstract behavioral models from environment implementations. In addition to relieving the user of the need to specify assumptions, by reifying abstract environment models as source code our approach can compactly express data-oriented assumptions that are difficult to express in the specification languages typically accepted by model checkers. Furthermore, the resulting models can be combined with the unit's implementation and submitted to existing program model checking frameworks to verify properties of the unit. Finally, the soundness of our static analyses and model generation eliminates the need to discharge environment assumptions.

In this paper we define modular program analysis and generation techniques that underly the modular program checking approach outlined in Figure 1. Users begin by identifying *unit properties* that characterize the correct behavior of a subsystem. Based on these properties a user manually identifies the set of classes, methods and fields that are relevant to the property and those form the unit. The environment is automatically identified and analyzed in two stages to determine its influence on unit data (i.e., objects whose type is a unit class). The first stage performs a scope-based analysis of the environment classes to determine, and eliminate, those classes and methods that cannot modify unit data. The second stage performs a modular flow-based *side-*

ESEC/FSE'03, September 1–5, 2003, Helsinki, Finland.

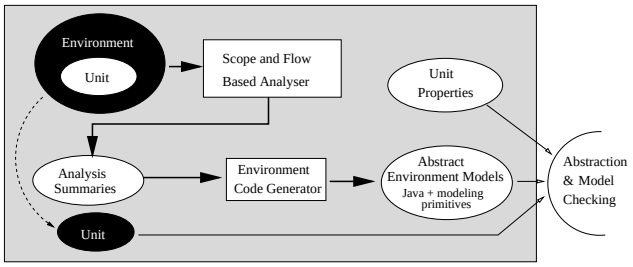

Figure 1: Analysis Framework Architecture

effects analysis for each environment method that calculates a safe, yet precise, *summary* of how the environment may modify the unit data. Analysis summaries drive the generation of Java code that references special *modeling primitives* designed to capture analysis approximations. These *code* models are combined with the unit class to form a legal *abstract* Java program that is amenable to abstraction and whose properties can be model checked using existing frameworks such as Bandera [4] and JPF [18].

Side-effects analyses have been widely used for optimization and software engineering applications (e.g. [1, 10]). These analyses typically calculate *may* side-effects information, which represents the set of non-local memory locations that are *possibly* modified on some execution of the method (e.g., definitions of global variables or fields of method parameters). In general, the side-effects of any individual method call are a subset of the may side-effects. Unfortunately, modeling each of those subsets can dramatically increase the cost (since the number of subsets grows exponentially with the total number of possible side-effects) and imprecision of model checking (since groups of side-effects are typically correlated). We address the problems with using may side-effects analysis results through the use of two refinements: *must* side-effects analysis, which calculates data modifications that occur on all executions of a method, and *return-sensitive* side-effects analysis, which calculates the set of side-effects for each exit point of a method.

For languages like Java, side-effects analysis is complicated by the presence of references, which allow indirect memory access. For example, in order to identify the data effects of the statement `l.f = r`, one needs to know the objects that `l` may refer to in order to conclude that the `f` field of those objects is modified. Side-effects analyses use the results of a *points-to* analysis, which approximates the set of objects pointed to by a reference variable. Approximation in points-to analysis leads to approximation in side-effects analysis. We address the need for precision by employing *flow and context-sensitive* points-to and side-effects analyses (e.g., [1, 10]) that take into account the order of statements in a method and gain a measure of context-sensitivity by calculating parameterized pointer information (e.g., [11]). Our analyses builds off of the approach of [10] and represents memory locations using *access path*s (i.e., chains of pointer references that a program may execute) that are limited to a maximum length k (i.e., are *k-limited*). We extend existing approaches to provide a means of tracking object types and whether an object is *reachable* via reference chains of length greater than k.

Finally, traditional side-effects analyses do not capture the possible values written into the side-effected location (e.g., for `l.f = r` they only calculate the possible values of `l.f`). We address this problem in our analysis by tracking assigned values in side-effecting statements (e.g., we calculate both the values of `l.f` and `r`). This leads to significantly more precise and efficient models for subsequent reasoning.

```
public class Subject{
 boolean changed = false;
 Buffer obs;
 public Subject() { obs = new Buffer(); }
 public synchronized void add(Watcher o)
  { obs.register(o);}
 public synchronized void delete(Watcher o)
  { obs.unregister(o);}
 public void notify(Object arg) {
  Watcher cw;
  Buffer lb = new Buffer();
  synchronized (this) {
   if (!changed) return;
   obs.copy(lb);
   changed = false;
  }
  if (obs.size() != lb.size())
    cw = null;
  while (!lb.isEmpty()) {
   cw = lb.removeFirst();
   cw.update(this, arg);}
 }
 protected synchronized void setChanged()
  { changed = true;}
}
public class Watcher{
  public int attempts = 0;
  public int aborts = 0;
  public boolean registered = false;
  public void update(Watcher o,
                     Object arg) { }
}
```

Figure 2: Custom Observer (excerpts)

In summary, our work makes several technical contributions, including (*i*) the adaptation of existing k-limited access path-based points-to analyses to exploit the partitioning of a program into unit and environment and provide a degree of precision for paths of length greater than k; (*ii*) the adaptation of existing side-effects analysis frameworks to model side-effecting values and to improve precision via *return sensitive* and *must* side-effects analyses; (*iii*) strategies for exploiting calculated data effects to generate safe abstract models of environment behavior; (*iv*) identification of model checker functionality required to support environment data modeling; and (*v*) a method for reifying abstract behavioral models as Java code that can be processed by Java abstraction and model checking tools. These analysis and environment generation techniques are modular (i.e., they consider only a single method at a time). Consequently, they scale effectively to large applications and can reduce large portions of an application to a compact model that is amenable to exhaustive analysis via model checking. The techniques have been implemented and applied to check properties of subsystems of large multi-threaded Java applications.

The next Section gives an overview of our basic approach. Section 3 describes our analysis framework for calculating environment data effect summaries. Section 4 describes how those summaries drive environment model generation. An overview of several case studies using our environment generation tools is presented in Section 5. Section 6 presents related work and Section 7 concludes by placing the contributions of this paper in the context of our broader tool-based methodology for environment generation.

```
public class Buffer extends Vector {
 public boolean register(Watcher w) {
  w.attempts++;
  if (!contains(w)) {
   w.registered = true;
   super.addElement(w);
   return true;
  }
  w.aborts++; return false;
 }
 public boolean unregister(Watcher w) {
  if (super.removeElement(w)) {
   w.registered = false;
   return true;
  } else return false;
 }
 public Watcher removeFirst() {
  Watcher result = elementAtIndex(0);
  removeElement(result);
  return result;
 }
}
```

```
public class Buffer {
 public static Buffer top = new Buffer();
 public Buffer newBuffer() { return this.top; }
 public boolean register(Watcher p0){
  if(choose()) p0.attempts = TOP_INT;
  if(choose()) p0.registered = true;
  if(choose()) p0.aborts = TOP_INT;
  return TOP_BOOL;
 }
 public boolean unregister(Watcher p0){
  if(choose()) p0.registered = false;
  return TOP_BOOL;
 }
 public Watcher removeFirst() {
  return choose("Watcher");
 }
}
```

Figure 3: Buffer Implementation and Generated Environment

2. OVERVIEW

Our analysis supports users who are interested in performing precise reasoning about correctness properties related to a known group of classes, fields and methods (i.e., the unit); for simplicity we restrict our presentation to units that consist of classes. We retain unit classes in the resulting system model, safely approximate the data effects of classes that interact with the unit classes, and safely eliminate classes and methods that do not effect data of unit type. In the remainder of this Section, we illustrate the main steps of this approach on a small publish-subscribe program. Figure 2 shows class `Watcher` whose instances observe changes to instances of class `Subject`. A field `obs` of type `Buffer`, shown on the left side of Figure 3, is a container for `Watcher`s that are registered for the `Subject`. The `Watcher` class contains bookkeeping fields that record the total number of registration `attempts`, the number of `aborts`, and whether the `Watcher` is `registered` on some `Subject`. Suppose, we are interested in reasoning about whether "Only registered `Watcher`s are notified of `Subject` updates". This can be specified in several ways, but one approach is to test whether the `registered` field of `Watcher`s is `true` at the point where a `Subject` calls `update()`.

2.1 Identifying the Unit

Unit selection is driven by the unit properties; classes mentioned in the properties should be included in the unit. The above mentioned property indicates that classes `Subject` and `Watcher` should be in the unit. In general, the unit consists of identified classes extended with fields and methods of super-type classes that are referenced by the methods of the identified classes. In our example, `Subject` and `Watcher` will be in the unit, but `Buffer` will be part of the environment.

2.2 Detecting Independent Classes/Methods

Even for relatively small Java systems, the environment may be very large and complex due to transitive class and method dependences. Scope-based information can be used to calculate an initial estimate of the classes and methods that cannot effect the unit data. Such methods can be omitted when building a procedure call graph, thus reducing the number of methods to be analyzed. For example, method `register()` calls `addElement()` of `java.util.Vector`. Since 1) none of the unit classes inherit fields from `java.util.Vector` and 2) class `java.util.Vector` has no knowledge of user-defined classes that make up the unit, then none of the methods of `java.util.Vector` can effect the unit data. Therefore, method `addElement()` can safely be excluded from further analysis.

```
public class Buffer {
 Watcher[] elementData;
 public void register(Watcher p0){
  if choose() elementData[TOP_INT] = p0; ...
 }
 public Watcher removeFirst(){
  return chooseReachable("Watcher",this);
 }
```

Figure 4: Generated Buffer *Container*

2.3 Analyzing Dependent Classes/Methods

A series of static analyses, including points-to and side-effects analyses, are applied to determine how the remaining methods of environment classes may influence the unit data. For the example, the analysis of the `register(Watcher w)` method in `Buffer` calculates that the assignment `w.registered = true` may effect the unit data.

2.4 Environment Properties

One of the hardest questions in environment modeling is "how much environment behavior may be ignored and what behavior should be preserved?". Rather than solve this problem in general, we have identified a property that occurs commonly for objects in Java programs: *containment*. An object *contains* another object if the latter is reachable through some chain of references from the former in a given program state. By treating selected environment fields (e.g., fields of `Buffer`) as part of the unit, our analyses can track data effects to such fields and generate more precise environment models. Such an analysis of `Buffer.removeFirst()` produces a summary that indicates that upon completion the method will return a `Watcher` that is reachable through the heap from the buffer instance (i.e., `this`). Figure 4 illustrates the use of non-determinism primitives (e.g., `chooseReachable("Watcher", this)`), discussed in the next section, to implement an abstraction of the `Buffer`'s containment relation.

2.5 Generating Environment Models

Models are generated to reflect all possible data effects as calculated by the preceding analyses. To safely reflect the *possibility* of a side-effect, code is generated to execute *abstract assignments* non-deterministically. This is achieved

by using a special modeling primitive, `choose()`, that is interpreted as either `true` or `false` by the model checker. The right side of Figure 3 shows the generated environment for `Buffer`. The assignment `w.registered = true` in method `register()` is approximated as the non-deterministic execution of assignment `w.registered = true` in the environment method to model the fact that according to the analysis results the assignment *may* cause a side-effect, but it may not.

Values assigned in statements with side-effects are also approximated. For scalars, constant values are preserved, as shown for `w.registered = true`, but for more complex expressions a $\top_t$ (denoted `TOP_t` in code) value is used. $\top_t$ represents all possible values of type t and the model checker is able to perform *abstract* calculations with such values [5]. For heap allocated data of unit classes, special modeling primitives are used to model the set of all instances of a class that are allocated in the current system state (e.g., `choose("Watcher")`). More precise modeling primitives are used to describe instances that are reachable from other instances (e.g., `chooseReachable("Watcher",this)`).

For environment classes, a per-type summary object is used to model all instances of a type (e.g., `Buffer top`). Unit statements that allocate, assign or compare values of environment data are modified to safely operate on these summary objects.

2.6 Abstraction and Model Checking

Model checking the example from Figure 2 with JPF using the environment model from Figure 3 yields a spurious counter-example where an unregistered `Watcher` is notified of an update. This is due to the imprecision of the generated `removeFirst()` which can return any allocated instance of type `Watcher`. Boosting the precision of the generated environment to model *containment* as shown in Figure 4 eliminates the spurious counter-example and reveals a race condition in the implementation of `notify()` that is due to the intentional limitation of the scope of the `synchronized` statement for improved performance.

3. DATA-EFFECTS ANALYSES

As described in Section 2, generated environment models are subjected to model checking. Given this we would like to generate environment models that can be parameterized by the information calculated during model checking to maximize the precision with which environment effects are approximated. To ensure safety of environments we must analyze all of the environment implementation to determine its potential effects on the unit data; this can be very expensive. We balance the tradeoff between the cost of analysis and precision of analysis results by employing a staged modular analysis.

3.1 Detecting Independent Classes/Methods

As the first stage of the analysis, we construct a call graph that is customized for subsequent analyses based on scope information extracted from environment methods. The scope-based analysis is a quick way to determine whether an environment method can access unit data without analyzing the actual code of the method. If the declared class of the environment method is a part of one package and the unit is a part of another package and these packages do not reference each other, then the environment method can not modify the unit data. For example, if the environment class is in a library and the unit contains only non-library classes that do not inherit fields from the environment classes, then the environment has no knowledge of the unit data. During the call graph construction, we filter out methods based on the criteria described above, and the resulting call graph is guaranteed to contain all methods that may effect the unit data. In the worst-case, the call graph is the same as the one that would be constructed by traditional methods; in practice, as discussed in Section 5, our side-effects preserving call graph can be orders of magnitude smaller. Despite its simplicity experience has shown that this analysis is very effective in pruning library code.

3.2 Analyzing Dependent Classes/Methods

Traditional side-effects analysis determines the set of memory locations that *may* be modified by some method execution. For object-oriented software, this requires points-to analysis to determine the set of objects that may be pointed by a reference. Points-to and side-effects analyses have been designed primarily to enable program optimization [1, 10]. For example, one may be interested in the locations that are not defined by a method so that values involving those locations can be safely reused across the method call. For such applications a *may* side-effect analysis that over-approximates the set of objects that are possibly side-effected by a method is appropriate.

We are also interested in calculating a safe approximation of method side-effects, but find that there are two deficiencies with existing approaches: (1) they do not approximate the values that are assigned in a side-effecting statement and (2) *may* side-effects results are very imprecise. The first problem is resolved by recording side-effects as pairs approximating the effected memory location and an approximation of the effecting value. Figure 3 illustrated the second problem where the environment method for `register()` allows several infeasible sets of side-effects (e.g., setting of `p0.attempts` may be skipped and setting `p0.registered` and `p0.aborts` may happen in the same method execution). May side-effects analysis results must be interpreted as defining the set of possible side-effects for all possible method executions. Since there may be method executions that only perform a subset of those side-effects, a safe environment model must reflect that possibility. Since the number of such subsets grows combinatorially with the number of individual side-effect statements, this can pose both performance and precision problems for subsequent analyses. Figure 5 illustrates two refinements of *may* side-effects analysis that lead to more compact and precise generated environment models. We can refine our environment generation by starting with *may* side-effects, factoring out the *must* side-effects that occur on all executions, and reifying them as unconditional assignments as shown in `registerMust()` of Figure 5 for `p0.attempts`. An additional refinement is to apply a simple form of path-sensitivity to distinguish side-effects resulting from paths exiting the method at different return statements. These *return sensitive* analysis results are achieved at no additional cost and can improve the precision of generated environments as shown in `registerReturnSensitive()` of Figure 5 where `p0.registered`'s side-effect is associated with return of `true`.

```
public boolean registerMust(Watcher p0) {
 p0.attempts = TOP_INT;
 if (choose()) p0.registered = true;
 if (choose()) p0.aborts = TOP_INT;
 return TOP_BOOL;
}
public boolean registerReturnSensitive(Watcher p0) {
 p0.attempts = TOP_INT;
 if (choose()) {
  if (choose()) p0.registered = true;
  return true;
 } else {
  if (choose()) p0.aborts = TOP_INT;
  return false;
 }
}
```

Figure 5: Side-effects Refinements

3.2.1 Program Representation

We describe our Java program analyses in terms of Jimple, a three-address representation of JVM byte-codes used in the Soot framework [17]. Our analyses proceed on a per-method basis. The variables, Var, accessed in a method consist of parameters, $p_i \in Var$, where p_0 refers to the receiver object, and locals, $l_i, r_i \in Var$. A class, $c \in Class$, has a set of associated fields, $f_i \in Field$, and methods, $m_i \in Method$; when not clear from the context we name a class' fields and methods explicitly as $c.f_i$ or $c.m_i$. We denote the classes identified as the unit as $U \subseteq Class$. We assume the presence of operators for accessing the type of expressions, $type(l_i)$, and for relating fields and methods to their containing class, $class(f)$. For convenience, we use f_U to denote the set of fields where $type(f) \in U$.

Points-to and side-effects analysis results are only dependent on assignment statements and method calls. Assignments in three-address form, $\texttt{l}_i\texttt{.f}_n\texttt{ = r}_j$, always refer to a local variable $\texttt{l}_i$ in forming the target address; we do not need to consider complex dereference expressions on the left-hand side of assignments. The following statements are treated by our analyses:

identity $\texttt{l}_i\texttt{ = p}_j$

allocation $\texttt{l}_i\texttt{ = new c}$

copy $\texttt{l}_i\texttt{ = r}_j$

load $\texttt{l}_i\texttt{ = r}_j\texttt{.f}_n$

store $\texttt{l}_i\texttt{.f}_n\texttt{ = r}_j$

invoke expression $\texttt{l}_i\texttt{ = r}_0\texttt{.m(r}_1\texttt{, ...,r}_n\texttt{)}$

invoke statement $\texttt{r}_0\texttt{.m(r}_1\texttt{, ...,r}_n\texttt{)}$

Our analysis treats array and field access expressions similarly, but for simplicity we limit our presentation to reference and scalar types.

3.2.2 Symbolic Locations

Fundamental to our analyses is our approach for representing the memory locations that a statement may reference. Our approach is based on length-limited access path based analyses (e.g., [10]) and symbolic analyses (e.g., [11]). We combine these approaches and adapt them to our setting in which the analysis distinguishes between unit data and environment data in order to precisely characterize points-to information for the former, but not the latter.

The goal of our analysis is different from the traditional goal of points-to analyses. In particular, we do not use analysis results to determine potential aliasing relationships and therefore do not require a *canonical* representation of points-to information. Our analyses are used to safely represent, at each program point, the state of the program, which maps variables to their values; values can be heap locations, a special value *null* denoting a null pointer, or scalar values (e.g., integers, reals, etc.).

Our points-to representation captures information relative to a given method and is parameterized by a *root symbol* that represents memory locations. There are three kinds of memory locations that may serve as a root: public static fields of classes (denoted $\widehat{c.f_i}$), method parameters (denoted $\widehat{p_i}$), and newly allocated data (denoted $\widehat{new_{c,s}}$ for class c allocated at statement s). New locations are modeled as per-allocator summary locations which are supported by the environment code generation described in Section 4. Our representation makes use of operations that denote sets of heap allocated objects in a given state. One can access the set of all allocated instances of class c (denoted $choose_c$), and the set of allocated instances of class c that are reachable from memory location l via paths through the heap that only reference unit data (denoted $reachUnit_C(l)$).

A *symbolic location* (denoted $sl \in \mathcal{SL}$) is a *null*, $choose_C$ expression or a length-limited access path (denoted π) of the form defined by this regular expression:

$$(\widehat{c.f_i} \mid \widehat{p_j} \mid \widehat{new_{c,s}}) f_U^{0-k} (reachUnit_C)?$$

An access path starts at a root symbol, consists of 0 to k dereferences of field accessors of unit type, and is optionally terminated in a reachable expression (where the parameter is understood to be the path prefix). We refer to a prefix of a path with j field dereferences as $\pi[j]$. The semantics of a path are defined relative to a program state, s. Paths represent field accesses that are *type correct* in the sense that $\pi[j]$ with type c can only be extended to length $\pi[j+1]$ by a field f where $class(f)$ is c. Paths end in either the location referred to by the field access sequence or a $reachUnit_C(\pi[k])$ expression. In the former case, the access path represents instances of class c that are reachable via the chain of field dereferences denoted by π in state s. In the latter case, the access path represents instances of class c that are reachable via a chain of field dereferences through unit data from any of the memory locations denoted by $\pi[k]$ in state s. Note that a variable, $l \in Var$, of a reference type may point to a set of symbolic locations, $S \in \mathcal{P}(\mathcal{SL})$, whose types are assignment-compatible, $sl \in S \Rightarrow type(sl) \leq type(l)$, where $\leq$ is the sub-typing relation.

Our symbolic locations provide a different degree of precision compared to traditional k-limited access path based representations (e.g., [10]) in that they are well-typed and they are able to represent heap reachability relationships between locations.

A pair of symbolic locations is ordered ($\leq$) based on the containment order of the sets of memory locations denoted by the pair. According to the semantics described above the order is:

$$\forall_{j \leq i} \quad \pi[i] \leq reachUnit_{type(\pi[i])}(\pi[j])$$
$$\forall_{c,i} \quad reachUnit_C(\pi[i]) \leq choose_c$$

This ordering is lifted to sets of symbolic locations as follows:

$$(\forall_{a \in S} \exists_{b \in S'} a \leq b) \rightarrow S \leq S'$$

A symbolic location can be *extended* by a field dereference (denoted $sl.f$) using the following rules: (a) if $sl = choose_{type(c)}$ then $sl.f = choose_{type(f)}$, (b) if $sl = \pi$ and

$type(f) \notin U$ then $sl.f = choose_{type(f)}$, and (c) for the remaining cases, we must consider the structure of π:

$$\pi.f = \begin{cases} \widehat{root} f_U^i f & \text{if } \pi = \widehat{root} f_U^i \wedge i < k \\ reachUnit_{type(f)}(\pi) & \text{if } \pi = \widehat{root} f_U^k \\ reachUnit_{type(f)}(\pi') & \text{if } \pi = reachUnit_c(\pi') \end{cases}$$

Symbolic locations can be *prefixed* (denoted $sl[[\bar{\pi}/\bar{p}]]$) by substituting the symbolic names of parameters, $p_i \in \bar{p}$, used in defining access path with type and length appropriate path prefixes, $\pi_i \in \bar{\pi}$. If a prefix operation causes the sequence of field dereferences to exceed k then the extension operator is applied for each field dereference beyond k. The intuition here is that we calculate a symbolic analysis summary for a method m and then use the extension operator (denoted $m[[\bar{a}/\bar{p}]]$) to determine the effects at a call site by substituting the actual parameters, $\bar{a}$, for the symbols representing the formal parameters, $\bar{p}$, of m.

Extension and prefixing operations can be lifted to sets of symbolic locations $S \in \mathcal{P}(\mathcal{SL})$ by extending or prefixing each constituent location (denoted $S.f$ and $S[[\bar{\pi}/\bar{p}]]$).

3.2.3 Points-to Analysis

This is a flow-sensitive, forward flow analysis. A set of points-to mappings from method locals to sets of symbolic locations (denoted $Pt : Var \rightarrow \mathcal{P}(\mathcal{SL})$) is calculated for entry and exit of each statement in the flow graph. An additional mapping $Pt_m : Method \rightarrow \mathcal{P}(\mathcal{SL})$ maps methods to their return locations as calculated by the points-to analysis at the exit point of the methods. The initial data flow set is empty. Sets are combined at flow-graph merge points by unioning the images of mappings with the same domain element.

Gen/Kill transfer functions are defined for assignment statements, s, mentioned above as follows:

$$\begin{aligned} Pt_{entry}(s) &= \bigcup\{Pt_{exit}(s') \mid s' \in pred(s)\} \\ Pt_{exit}(s) &= (Pt_{entry}(s) - Kill(s)) \cup Gen(s) \end{aligned}$$

For clarity we define $Locs(l) = S \mid (l \rightarrow S) \in Pt_{entry}(s)$, the set of locations that l points to at the entry point of statement s.

The identity function is used for all statements that do not assign a reference variable. For the remaining statements, the $Kill$ sets are of the form:

$$\begin{aligned} Kill(\texttt{l = ...}) &= \{l \rightarrow S \mid (l \rightarrow S) \in Pt_{entry}(s)\} \\ Kill(\texttt{l.f = r}) &= \{x \rightarrow sl \mid (x \rightarrow sl) \in Pt_{entry}(s) \wedge \\ &\qquad \exists k \mid sl[k] \in Locs(l).f\} \end{aligned}$$

The $Kill$ function for the store statement calculates all references x that point to a location sl whose access paths contain the heap reference f that gets modified by the statement. As a safe approximation, such variables will point to $choose_{type(x)}$ after the statement.

For statements whose assigned type is not in the unit the Gen set is:

$$Gen(\texttt{l = ...}) = \{l \rightarrow choose_{type(l)}\}$$

In all other cases the Gen sets for statement, s, are:

$$\begin{aligned} Gen(\texttt{l = p}) &= \{l \rightarrow \{p\}\} \\ Gen(\texttt{s}_i\texttt{: l = new C}) &= \{l \rightarrow \{new_{c,i}\}\} \\ Gen(\texttt{l = c.f}) &= \{l \rightarrow \{c.f\}\} \\ Gen(\texttt{l = r}) &= \{l \rightarrow Locs(r)\} \\ Gen(\texttt{l = r.f}) &= \{l \rightarrow Locs(r).f\} \\ Gen(\texttt{l.f = r}) &= \{x \rightarrow choose_{type(x)} \mid \\ &\qquad (x \rightarrow sl) \in Kill(s)\} \\ Gen(\texttt{l = r}_0\texttt{.m(r}_1\texttt{,...)}) &= \{l \rightarrow Pt_m(m)[[\bar{Locs}(r_i)/\bar{p_i}]]\} \end{aligned}$$

where $Pt_m(m)$ is a set of return locations as calculated by the points-to analysis for m.

3.2.4 Side-Effects Analysis

Side effects occur in **store** statements of the form: $\texttt{l}_i\texttt{.f}_n\texttt{ = r}_j$ Our side-effects analysis uses the symbolic locations calculated for $\texttt{l}_i$ at an assignment statement to determine the set of objects whose fields may be referenced as the target of the assignment. The value of the right-hand side of such an assignment is also safely approximated by looking up the symbolic values referenced by $\texttt{r}_j$.

As mentioned previously, we calculate both *may* and *must* side-effects information. These are flow-sensitive, forward flow analyses. The analyses relate side-effected symbolic locations to sets of *symbolic values*, $\mathcal{SV} = \{\mathcal{SL} \cup Scalar\}$. $Scalar$ is the domain of values for all non-reference variables lifted to contain a $\top_t$ value, for each type t, that represent all possible values of type t; the values in $Scalar$ are similar to values in a constant propagation lattice [13], however, our analysis can keep track of a set of constant values. A set of side-effects mappings from symbolic locations to sets of symbolic values (denoted $Se^{may}, Se^{must} : \mathcal{SL} \rightarrow \mathcal{P}(\mathcal{SV})$) is calculated for entry and exit of each statement in the flow graph. Sets are combined at flow-graph merge points by unioning domain values with the same $\mathcal{SL}$ elements for *may* analysis and by intersecting them for *must* analysis. The initial data flow sets for *may* and *must* analyses are empty.

Transfer functions are defined for **store** and **invoke** statements, s, as follows:

$$\begin{aligned} Se^{may}_{entry}(s) &= \bigcup\{Se^{may}_{exit}(s') \mid s' \in pred(s)\} \\ Se^{may}_{exit}(s) &= (Se^{may}_{entry}(s) - Kill^{may}(s)) \cup Gen^{may}(s) \\ Se^{must}_{entry}(s) &= \bigcap\{Se^{must}_{exit}(s') \mid s' \in pred(s)\} \\ Se^{must}_{exit}(s) &= (Se^{must}_{entry}(s) - Kill^{must}(s)) \cup Gen^{must}(s) \end{aligned}$$

A set of symbolic values is denoted $V \in \mathcal{P}(\mathcal{SV})$. Symbolic value *prefixing* (denoted $sv[[\bar{\pi}/\bar{p}]]$) is defined analogously to prefixing for symbolic locations except that the identity function is used for scalar values. For clarity we define $Vals(r) = Locs(r) \cup Scalars(r)$, where $Scalars(r)$ is defined for a scalar type r and returns a set of scalar values r may be assigned to. $Kill$ sets are defined for **store** statements:

$$\begin{aligned} Kill(\texttt{l.f = r}) &= \{sl.f \rightarrow V \mid sl \in Locs(l) \wedge \\ &\qquad (sl.f \rightarrow V) \in Se_{entry}(s)\} \end{aligned}$$

Gen sets for **store** and **invoke** statements, s, are defined

as:

$$\begin{aligned} Gen(\texttt{l.f = r}) &= \{sl.f \rightarrow Vals(r) \mid sl \in Locs(l)\} \\ Gen(\texttt{r}_0\texttt{.m(r}_1,\ldots\texttt{)}) &= \{sl[[\bar{Locs}(r_i)/\bar{p_i}]] \rightarrow V[[\bar{Locs}(r_i)/\bar{p_i}]] \mid \\ &\quad (sl \rightarrow V) \in Se_m)\} \end{aligned}$$

where Se_m denotes either the *must* or *may* analysis summary for m. For all other statements the identity transfer function is used.

Calculating *must* side-effects relies on *may Pt* information. To incorporate that information safely, the $Gen/Kill$ functions for *must* side-effects analysis are defined as for *may* analysis except for store statements, `l.f = r`. In that case, if variable `l` may point to more than one symbolic location, then Gen returns the empty set. This is safe because if may points-to analysis calculates that `l` may point to a single location, then it must point to exactly one location. To see this consider the case where the may points-to analysis calculates that at state s, the variable `l` points-to one symbolic location sl. If there is another path leading to s, then `l` is either *null*, sl, or is assigned to another value sl' on that path. $Locs(l)$ will have size one only if `l` is assigned the same value on all paths leading to state s, thus that singleton points-to information can be safely used to calculate must side-effects information.

3.2.5 Return-Sensitive Side-Effects Analysis

For methods with multiple return points, due to the flow-sensitive nature of our analysis there may be different side-effects summaries calculated at each method return point. Rather than merge those sets to produce a single summary of side-effects for the method, we produce a side-effects summary for each return point and consider the method summary as the set of those summaries.

3.3 Safety of Side-Effects Analysis Results

To prove the correctness of points-to/side-effects analysis we define a simulation relation between the true state of the program, as recorded by the program's semantics, and the abstract information about the state of the program, as recorded by points-to/side-effects analysis. We prove that the simulation relation holds in [15].

4. GENERATING ENVIRONMENT MODELS

In this section, we describe how special modeling primitives may be used to generate environment abstract models from the analysis summaries.

4.1 Modeling Environment Data

By design the analyses of the preceding section intentionally ignore any differences between instances of objects of environment types. To minimize the state space of the environment, for each environment type our models store a single object instance that *summarizes* the state of all concrete instances. Every allocation of an environment type is transformed to a call to a method, for example `newBuffer()` in Figure 3, that returns a reference to the single class summary instance. Field values are approximated by using $\top_t$ values for each *summary* instance field; we note that for reference fields a $\top_C$ has the semantics of $choose_C$. Object identity (i.e., reference value) is used in object equality comparisons. Where these occur in unit code a transformation is necessary that replaces the `==` expression with a call to an `equals()` method that uses `choose()` to reflect the inability of environment summary instance to distinguish identity. For the example in Figure 2, the placement of `Buffer` in the environment requires the following definition of object equality:

```
public static boolean equals(Object x, Object y){
 if(x instanceof Buffer || y instanceof Buffer)
  return choose();
 else
  return x == y;
}
```

4.2 Modeling Primitives

Reifying analysis results as environment models encoded as Java program fragments requires primitives for expressing the approximations that naturally arise in points-to and side-effects analyses. Our approach to environment generation is model checker independent to a great extent, but it does require model checking framework to support these primitives; currently JPF and Bandera provide such support.

We define *modeling primitives* that capture the primitives used in defining symbolic locations. We introduce non-deterministic choice primitives over heap allocated data, `choose("C")` and `chooseReachable("C",l)` where `l` is any object reference expression. The semantics of these primitives correspond to the meaning of $choose_C$ and $reachUnit_C(l)$ from Section 3.

4.3 From Side-Effects to Code

Environment code generation is based on the analysis summaries from the previous section. For each environment method that may be invoked by the unit, we generate an environment method and its enclosing class.

Side-effecting statements are modeled as *abstract assignments* that may *write* to a set of unit locations. The written locations are elements of $\mathcal{SL}$. There are three possibilities for such values: (1) the location is $choose_C$ in which case `choose("C")` is emitted; (2) the location is $p_i f_1 f_2 \ldots f_k$ in which case the unit expression, e, that is passed as parameter p_i is used to expand the symbolic location to emit $e.f_1.f_2...f_k$; and (3) the location is $p_i f_1 f_2 \ldots f_k reachUnit_C$ in which case the unit expression, e, that is passed as p_i is used to expand the symbolic location to emit `chooseReachable("C",` $e.f_1.f_2...f_k$). Note that for access paths rooted at new locations and globals no path extension is required. *Scalar* values in side-effect statements are emitted either as the appropriate literal value or as enumerated `TOP` values whose semantics are recognized by the model checking framework. The resulting assignments may have non-determinism on both their left and right-hand sides. Model checkers that support the required modeling methods will generate each possible pair of left and right-hand side values in order to safely approximate all possible assignments.

For *must* side-effects, assignments are guaranteed to occur in any method execution, thus the abstract assignments are included directly in the environment method body. For *may* side-effects, assignments can possibly occur in any method execution, thus the abstract assignments are included in the consequent of a conditional statement with `choose()` as the condition expression. For *return-sensitive* side-effects, the summary is a set of sets of side-effects. We generate a chain

of conditionals each guarded by `choose()` such that the conditional bodies are mutually disjoint and each of the abstract assignments for each set of side-effects is placed in its own body. Figure 5 illustrates the three code generation cases.

Since environment code generation is essentially a direct encoding of side-effects analysis results, the safety of those environments follows from the safety of the preceding analyses and the semantics of the modeling primitives.

5. EXPERIENCE WITH TOOLS

We have applied our tools to several large Java applications to assess the scalability of our techniques and to several smaller programs that use collection data structures from libraries to assess the tool's ability to identify *containment* properties of the environment.

For many of these programs, the primary benefit of the environment analysis and generation tools is to eliminate library code from the system that is subsequently model checked. While this is clearly useful, one might question the need for the sophistication of our approach when one can simply assume that library code can be "ignored" as is done in many other program analysis tools. We contend that a general analysis capability like ours not only handles pre-defined library code in a rigorous way, but it allows for arbitrary parts of the program to be treated as *libraries* and abstracted as part of the environment.

The model checks of the real programs, discussed below, involved local properties of individual classes or small tightly-coupled groups of classes. The environment classes in those programs had few side-effects which resulted in generated environments that were simple in terms of their structure, however, they were large enough to make manual environment definition a tedious and error prone task. In contrast, the examples that used collections needed the more sophisticated analysis framework to generate sufficiently precise models. Clearly more experience with our environment generation tools is needed to understand the degrees of control that users may require in adjusting the precision of generated environments.

5.1 Replicated Workers

Replicated workers is a configurable framework designed to support the parallelization of simulations. The replicated workers architecture includes a shared pool and a number of workers that repeatedly access the data from the pool, perform the computation, and put the new data back to the pool. The job is finished when the work pool becomes empty. The user can specify the number of workers executing concurrently and several other attributes of their collaborative execution. We studied an application of the replicated workers that used it to solve a standard Jacobi relaxation problem. The goal was to extract the replicated workers framework from the application while safely modeling the effects of the framework. Scope-based analysis was very effective; it reduced the number of classes that had to be considered by side-effects analysis from 431 to 31. The resulting unit was comprised of 6 classes and approximately 500 lines of code; the generated environment consisted of 3 classes and very little code since it was determined that the application caused no side-effects on the framework except for boolean return values.

We checked several properties from [6] using both Bandera and got the same results as in that study. In one case, however, our results diverged from what we expected. When checked a framework instance for deadlock, we found an actual deadlock. The bug was in the implementation of a barrier synchronization utility. Its discovery was surprising since the framework has been used in implementing more than ten non-trivial parallel simulation applications and this bug was never discovered. We replaced the barrier implementation with one from `java.util.concurrent` and the deadlock was eliminated.

5.2 Flight Simulator

We analyzed the implementation of a flight simulator's cockpit display. The autopilot tutor is a web-based application that has a GUI for simulating the Autopilot Mode Control Panel and a Primary Flight Display of an MD-11 aircraft autopilot. A user may click on buttons to dial desired altitude and vertical speed, and advance the aircraft towards its goal altitude. Autopilot is implemented as an applet. It uses GUI toolkits such as `java.awt` and `java.swing` to paint the interface on a screen.

We checked the autopilot simulator implementation for *mode confusions*. Mode confusions in systems with human-machine interaction are scenarios when the operator thinks that the machine is in one mode and the machine is in a different mode. Human-machine interaction systems are complicated by presence of multiple agents in the system: the user (pilot), the task ("take the aircraft to a certain altitude"), the machine (the autopilot), and the interface between the operator and the machine (knobs, wheels, and displays in a cockpit).

In order to find mode confusions related to altitude deviation errors we hand-coded a simple model of a user's beliefs, generated environment methods for all the GUI components, applied integer abstraction to selected system objects, and generated drivers according to specific pilot task expressed as regular expressions. We restrict our attention here to the generation of the environment methods, for a more complete description see [16].

The main class of the applet is `Autopilot` which extends `java.applet.Applet` which in turn extends several `AWT` classes. This applet makes a large number of calls to `AWT` methods in order to create and update the simulated cockpit displays. The `Autopilot` class is over 3,500 lines of dense code, mainly GUI related. The properties we wished to reason about were independent of the state of the GUI and we were free to choose the `Autopilot` class itself as the system under analysis.

Scope based analysis was essential in enabling the more expensive side-effects analyses to run. The number of classes was reduced from 1474 to 41. The side-effects analysis determined that there were several side-effects on explicitly defined fields of `Autopilot` and on fields inherited from `AWT` classes. The resulting environment consisted of 15 classes that were generated.

Submitting the user-model, autopilot implementation, the pilot actions program and generated environment to JPF resulted in finding a mode confusion scenario in a matter of minutes. No manual work was required to deal with the massive GUI libraries in this example.

5.3 Container Examples

We studied a group of programs that use library components such as `List`, `Vector` and a binary search tree (

```
public class BSTree {
 BSTNode rootNode;
 int elementCount;
 public void put(SearchKey keyObj, Leaf value) {
  SearchKey s = keyForObject(keyObj);
  BSTNode n = search(s);
  if (n == null) {
   n = new BSTNode(s, value);
   insert(n);
  }
 }
 private void insert(BSTNode n) {
  BSTNode y = null;
  BSTNode x = rootNode;
  while (x != null) {
   y = x;
   if (n.key.compareKey(x.key) < 0)
    x = y.left;
   else x = y.right;
   n.parent = y;
   if (y == null) rootNode = n;
   else if (n.key.compareKey(y.key) < 0)
    y.left = n;
   else y.right = n;
  }
  elementCount++;
 }
```

```
// Unit code referencing BSTree
SearchKey k = new SearchKey();
Leaf l = new Leaf();
BSTree bst;
...
bst.put(k,l);

// Side-effects summary
public Leaf put(SearchKey p0, Leaf p1){
 must: this.elementCount = TOP_INT;
 may: reachable("Leaf", this) = p1;
 may: reachable("SearchKey", this) = p0;
}

// Contains-preserving environment
public class BSTree {
 Object[] contains;
 public Leaf put(SearchKey p0, Leaf p1){
  if (choose()) contains[choose(0,contains.length)] = p1;
  if (choose()) contains[chooseInt(0,contains.length)] = p0;
  this.elementCount = Abstraction.TOP_INT;
 }
 public Leaf query(SearchKey p0){
  if (choose()) return reachable("Leaf",this);
  else return null;
 }
```

Figure 6: Search Tree, Client and Environments (excerpts)

`BSTree`). We analyzed these components to identify *containment* properties that can drive the generation of more precise environment models.

Consider Figure 6 which shows the `put()` method of a binary search tree implementation on the left hand side. This implementation uses `keyObj` as a key to store `value` into the tree. The `insert()` method navigates down to a leaf of the tree and links the newly created `BSTNode` instance that holds the key and value into an existing leaf node.

We configured the side-effects analysis to treat fields of `BSTree` and `BSTNode` as part of the unit. This allowed the analysis to track access paths through the search tree nodes and the results indicated that after executing the `put()` method, key and value will be *reachable* from the binary search tree root. The side-effects analysis results for methods `remove(SearchKey keyObj)` and `query(SearchKey keyObj)` of the binary search tree yield similar results.

Rather than generate code as described in Section 4.3, we instead generate an abstract model for the analyzed classes that are directly called from the unit, in the case of our example this is `BSTree`. This means that all instances of `BSTNode` will be eliminated from the system. This appears problematic because access paths that traverse those nodes make the `Leaf` nodes reachable from the root (i.e., `this`). We can generate an environment model for `BSTree` that preserves an abstract *containment* relation, even with the elimination of `BSTNode`s, by associating an array with instances of the search tree. Assignment to a `reachable(type,this)` expression is converted into a slot in the array and since the array is a field of the environment class its elements can be accessed (non-deterministically) by `reachable(type,this)` expressions. The resulting environments can yield state space reductions since all of the interior nodes of a tree are eliminated and replaced with an array of references to the leaves of the tree.

These same techniques were applied to the containers in the Replicated Workers example although they were not necessary to reveal the deadlock in that program.

6. RELATED WORK

The problem of detecting inter-procedural side-effects in the presence of pointers has been widely studied. Traditionally such analyses have been used to enable program optimizations (e.g., [1, 10]) more recently researchers have considered targeting software engineering client applications with sophisticated side-effects analyses [14].

The most common technique for calculating method side-effects is to calculate points-to information first and then perform side-effects analysis. There are several approaches to detecting aliases in the literature, but as discussed previously, our work builds off of access path approaches [10]. An important distinguishing feature of our work is that when our approximation reaches its bound we can distinguish locations that are reachable from a k-limited path from the set of all locations. In addition, we exploit type information to distinguish sets of reachable locations.

Our analyses draw various aspects from other approaches. Our analysis is modular, like [19, 1], flow-sensitive, like [1, 10], and gains a measure of context-sensitivity by calculating parameterized pointer information, like [11]. Unlike existing analyses, ours distinguishes between unit and environment locations. Side-effects to environment locations are ignored, whereas side-effects to unit locations are kept tracked. In addition our analysis keeps track of not only what locations may be modified but what values they may hold at the end of a method's execution. Finally, based on our client application we combine may, must and return-sensitive information to produce a more precise characterization of a methods side-effects.

Verisoft incorporates a tool that calculates the influence of externally defined data on the system under analysis [3]. Unlike in our approach, they use a simple notion of data dependence to drive their analysis and do not have the ability to control the precision of the generated environments.

We note that there has been recent work on environment generation that takes a different approach. The basic idea is to infer a suitable environment assumption given unit and the property under consideration [2, 7]. To date these methods have not considered data interactions between the unit and environment. It would be worthwhile exploring the degree to which our techniques could be used to extend those methods.

7. CONCLUSIONS

The research presented in this paper is part of a larger project related to modular program model checking. The Bandera Environment Generator is a system that incorporates both environment *extraction* and environment *synthesis*. In this paper, we described how data effects are treated in environment extraction. Control effects can also be extracted from applications via control flow analysis. The tools are currently configured to assume the lack of divergence, indefinite-blocking, and lock acquisition in the environment. Ongoing work is directed at developing static analyses to check those assumptions.

In this paper we have described the combination and adaptation of several approaches to points-to and side-effects analyses. Our approach was targeted at a specific software engineering client application and as such several of the classic assumptions of existing analyses were not appropriate. We have defined a flexible analysis framework that can be tuned by the user to control the degree of precision it admits and developed code generation strategies that exploit analysis results to produce models that support efficient program model checking. Our experience to date suggests that our staged analysis approach can scale to large systems and can enable checking of properties of Java applications that were not previously possible.

8. ACKNOWLEDGMENTS

The authors would like to thank Torben Amtoft for his generous help with the analysis and the members of the Bandera and JPF projects, especially Robby and Corina Păsăreanu, for their comments. This work was supported in part by the U.S. Army Research Office (DAAD190110564), by DARPA/IXO's PCES program (AFRL Contract F33615-00-C-3044), and by Intel Corporation (Grant 11462).

9. REFERENCES

[1] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245, Charleston, South Carolina, 1993.

[2] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning assumptions for compositional verification. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (LNCS 2619)*, 2003.

[3] C. Colby, P. Godefroid, and L. J. Jagadeesan. Automatically closing open reactive programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 345–357, 1998.

[4] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera : Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.

[5] M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Păsăreanu, Robby, W. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering*, May 2001.

[6] M. B. Dwyer and C. S. Păsăreanu. Filter-based model checking of partial systems. In *Proceedings of the Sixth ACM SIGSOFT Symposium on Foundations of Software Engineering*, Nov. 1998.

[7] D. Giannakopoulou, C. S. Păsăreanu, and H. Barringer. Assumption generation for software component verification. In *Proceedings of the 17th IEEE Conference on Automated Software Engineering*, 2002.

[8] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, pages 58–70, Jan. 2002.

[9] O. Kupferman and M. Y. Vardi. Modular model checking. In *COMPOS (LNCS 1536)*, 1998.

[10] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural side effect analysis with pointer aliasing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 56–67, 1993.

[11] D. Liang and M. J. Harrold. Efficient computation of parameterized pointer information for interprocedural analyses. In *Proceedings of the 8th International Static Analysis Symposium (LNCS 2126)*, page 279, 2001.

[12] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1991.

[13] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[14] A. Rountev and B. G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *Proceedings of the 10th International Conference on Compiler Construction (CC'01)*, 2001.

[15] O. Tkachuk. Adapting side effects analysis for modular program model checking. Master's thesis, Kansas State University, 2003.

[16] O. Tkachuk, G. Brat, and W. Visser. Using code level model checking to discover automation surprises. In *Proceedings of the 2002 Digital Avionics Systems Conference*, 2002.

[17] R. Valle-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proceedings of CASCON'99*, Nov. 1999.

[18] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proceedings of the 15th IEEE Conference on Automated Software Engineering*, Sept. 2000.

[19] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, pages 187–206, Denver, CO, Oct. 1999. ACM Press.