Dynamic Package Interfaces

Shahram Esmaeilsabzali Rupak Majumdar
MPI-SWS
{*shahram,rupak*}*@mpi-sws.org*

Thomas Wies
NYU
wies@cs.nyu.edu

Damien Zufferey
IST Austria
zufferey@ist.ac.at

***Abstract*—Programmers using a package must follow protocols that specify when it is legal to call particular methods with particular arguments. For example, one cannot use an iterator over a set once the set has been changed directly or through another iterator. We formalize the notion of *dynamic package interfaces* (DPI), which generalize state-machine interfaces for single objects studied previously, and give an automatic algorithm to compute a sound abstraction of a dynamic package interface. States of a DPI represent sets of heap configurations, and edges represent the effects of method calls on the heap. Technically, we introduce a novel heap analysis to deal with potentially unboundedly many object instances and their interrelations, extending shape analysis with ideal abstractions for depth-bounded systems. Our algorithm performs abstract reachability analysis over this (infinite-state) abstraction, but is guaranteed to converge. We have implemented our algorithm for a Java-like source language, and show that our algorithm is effective in computing representations of common patterns of package usage, such as relationships between viewer and label, container and iterator, and JDBC statements and cursors.**

I. Introduction

Modern object-oriented programming practice uses packages to encapsulate components, allowing programmers to use these packages through well-defined application programming interfaces (APIs). While programming languages such as Java or C# provide a clear specification of the *static* APIs of components in terms of classes and their (typed) methods, there is usually no specification of the *dynamic* behavior of packages that constrain the temporal ordering of method calls on different objects. For example, one should invoke the *lock* and *unlock* methods of a lock object in alternation; any other sequence raises an exception. More complex constraints connect method calls on objects of different classes. For example, in the Java Database Connectivity (JDBC) package, a ResultSet object, which contains the result of a database query executed by a Statement object, should first be closed before its corresponding Statement object can execute a new query.

In practice, such temporal constraints are not formally specified, but explained through informal documentation and examples, leaving programmers susceptible to bugs in the usage of APIs. Being able to specify dynamic interfaces for components that capture these temporal constraints enable programmers to write client code for a package that observes the constraints imposed by the package. Moreover, program analysis tools may be able to automatically check whether the client code invokes the component correctly according to such an interface.

Previous work on mining dynamic interfaces through static and dynamic techniques has mostly focused on the single-object case (such as a lock object) [15], [2], [8], [7], [6], and rarely on more complex collaborations between several different classes (such as JDBC clients) interacting through the heap [10], [12]. In this paper, we propose a systematic, static approach for extraction of dynamic interfaces from existing object-oriented code.

More precisely, we work with *packages*, which are sets of classes. A configuration of a package is a concrete heap containing objects from the package as well as references among them. A *dynamic package interface* (DPI) specifies, given a history of constructor and method calls on objects in the package, and a new method call, if the method call can be executed by the package without causing an error. In analogy with the single-object case, we are interested in representations of DPIs as finite state machines, where states represent sets of heap configurations and transitions capture the effect of a method call on a configuration. Then, a method call that can take the interface to a state containing erroneous configurations is not allowed by the interface, but any other call sequence is allowed.

The first stumbling block in carrying out this analogy is that the number of states of an object, that is, the number of possible valuations of its attributes, as well as the number of objects living in the heap, can both be unbounded. As in previous work [7], [13], we can bound the state space of a single object using *predicate abstraction*, that tracks the abstract state of the object defined by a set of logical formulas over its attributes. However, we must still consider unboundedly many objects on the heap and their interrelationships. Thus, in order to compute a dynamic interface, we must answer the following questions.

1) The first challenge is to define a finite representation for possibly unbounded heap configurations and the effect of method calls. For single-object interfaces, states represent a subset of finitely-many attribute valuations, and transitions are labeled with method names. For packages, we have to augment this representation for two reasons. First, the number of objects can grow unboundedly, for example, through repeated calls to constructors, and we need an abstraction to represent

unbounded families of configurations. Second, the effect of a method call may be different depending on the receiver object and the arguments, and it may update not only the receiver and other objects transitively reachable from it, but also other objects that can reach these objects.

2) The second challenge is to compute, in finite time, a dynamic interface using the preceding representation. For single-object interfaces [2], [7], interface construction reduces (roughly) to abstract reachability analysis against a most general client (a program that non-deterministically calls all available methods). For packages, it is not immediate that abstract reachability analysis will terminate, as our abstract domains will be infinite, in general.

We address these challenges as follows.

Our first contribution is a novel shape domain for finitely representing infinite sets of heap configurations. Our shape domain is motivated by the insight that the objects instantiated by a package form heaps that can be naturally described as recursive unfoldings of nested graphs. Technically, our shape domain combines predicate abstraction [13], [11], for abstracting the internal state of objects, with a result on the finite representation of sets of depth-bounded graphs as nested graph structures [16].

To compute the dynamic package interface, we apply a general result on reachability of depth-bounded graph rewriting systems [17] to obtain a finite state abstraction of a program that consists of the package and its universal client. Our second contribution is an algorithm to extract the DPI from this finite state abstraction. We use the insight that the finite state abstraction can be reinterpreted as a numerical program. The analysis of this numerical program yields detailed information about how a method affects the state of objects when it is called on a concrete heap configuration, and how many objects are effected by the call.

We have implemented our algorithm on top of the Picasso abstract reachability tool for depth-bounded graph rewriting systems. We have applied our algorithm on a set of standard benchmarks written in a Java-like OO language, such as container-iterator, JDBC query interfaces, etc. In each case, we show that our algorithm produces an intuitive DPI for the package within a few seconds.

II. Overview: A Motivating Example

We illustrate our approach through a simple example.

Example Figure 1 shows two classes Viewer and Label in a package, adapted from [10], and inspired by an example from Eclipse's ContentViewer and IBaseLabelProvider classes. A Label object throws an exception if its run or dispose method is called after the dispose method has been called on it. There are different ways that this exception can be raised. For example, if a Viewer object sets its f reference to the same Label object twice, after the second call to set, the Label object, which is already disposed, raises an exception. As another example, for two Viewer objects that have their f reference attributes point to the same Label object, when one of the objects calls its done method, if the other object calls its done method an exception will be raised. An *interface* for this package should provide possible configurations of the heap when an arbitrary client uses the package, and describe all usage scenarios of the public methods of the package that do not raise an exception.

Dynamic Package Interface Intuitively, an interface for a package summarizes all possible ways for a client to make calls into the package (i.e., create instances of classes in the package and call their public methods). In the case of single-objects, where all attributes are scalar-valued, interfaces are represented as finite-state machines with transitions labeled with method calls [15], [2], [7]. Each state s of the machine represents a set $[\![s]\!]$ of states of the object, where a state is a valuation to all the attributes. (In case there are infinitely many states, the methods of [2], [7] abstract the object relative to a finite set of predicates, so that the number of states is finite.) An edge $s \xrightarrow{m} t$ indicates that calling the method $m()$ from any state in $[\![s]\!]$ takes the object to a state in $[\![t]\!]$. Some states of the machine are marked as errors: these represent inconsistent states, and method calls leading to error states are disallowed.

Below, we generalize such state machines to packages.

States: Ideals over Shapes The first challenge is that the notion of a state is more complex now. First, there are arbitrarily many states: for each n, we can have a state with n instances of Label (e.g., when a client allocates n objects of class Label; moreover, we can have more complex configurations where there are arbitrarily many viewers, each referring to a single Label, where the Label may have disposed = true or not. We call sets of (potentially unbounded) heap configurations *abstract heaps*.

Our first contribution is a novel finite representation for abstract heaps. We represent abstract heaps using a combination of *parametric shape analysis* [13] and *ideal abstractions for depth-bounded systems* [17]. As in shape analysis, we fix a set of unary predicates, and abstract each object w.r.t. these predicates. For example, we track the predicate `disposed`(l) to check if an object l of type Label has disposed set to true. Additionally, we track references between objects by representing the heap (roughly) as a nested graph whose nodes represent predicate abstractions of objects and whose edges represent references from one object to another. Unlike in parametric shape analysis, references are always determinate and the abstract domain is therefore still infinite.

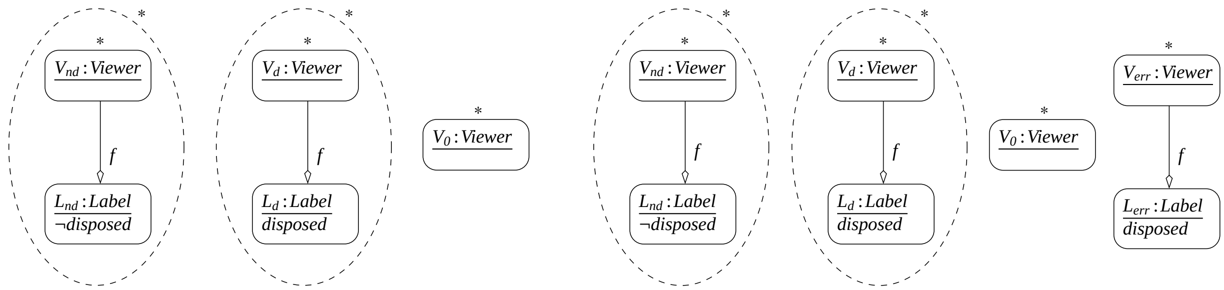
Figure 1(c) shows an abstract heap H_0 for our example. There are five nodes in the abstract heap. Each node has a name, a type, and a valuation of predicates, and represents an object of that type whose state satisfies the predicates (e.g., V_{nd} : *Viewer* represents a Viewer object and L_d : *Label*

```
class Viewer {
  Label f;
  public void Viewer() { f := null; }
  public void run() { if (f != null) f.run(); }
  public void done() { if (f != null) f.dispose(); }
  public void set(Label l){
    if (f != null) f.dispose();
    f := l; }
}
```

(a) The Viewer class

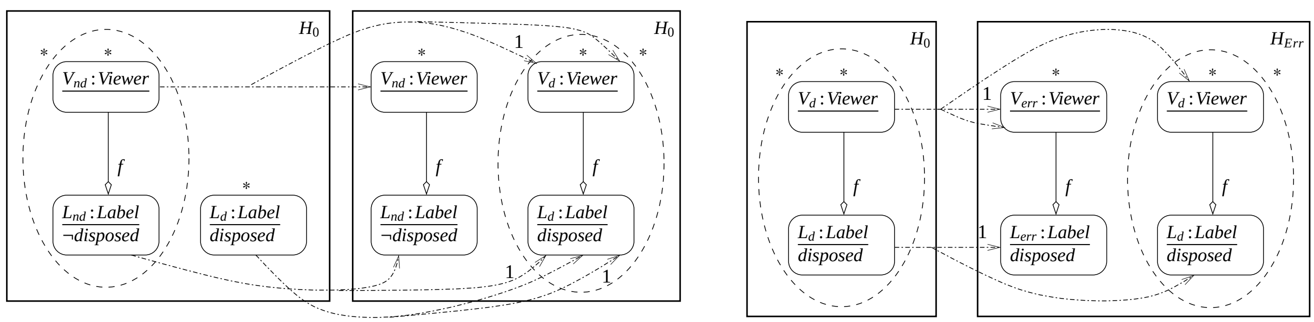
```
class Label {
  boolean disposed;
  public void Label() { disposed := false; }
  protected void run() {
    if (disposed) throw new Exception(); }
  protected void dispose() {
    if (disposed) throw new Exception();
    disposed := true; }
}
```

(b) The Label class



(c) Abstract heap configuration H_0

(d) Abstract heap configuration H_{Err}



(e) Object mapping for V_{nd} : *Viewer*.set(L_d : *Label*)

(f) Object mapping for : V_d : *Viewer*.set($*$: *Label*)

Figure 1. A package consisting of Viewer and Label classes together with its interface

represents a Label object for which disposed is true). Edges between nodes show field references: the edge between V_d and L_d labeled f shows that objects represented by V_d have an f field referring to some object in L_d. Finally, nodes and subgraphs can be marked with a "*". Intuitively, the "*" indicates an arbitrary number of copies of the pattern within the scope of the "*". For example, since V_d is starred, it represents arbitrarily many (including zero) Viewer objects sharing the Label object L_d. Similarly, since the subgraph over nodes V_d and L_d is starred, it represents configurations with arbitrarily many Label objects, each with (since V_d is starred as well) arbitrarily many viewers associated with it.

Figure 1(d) shows a second abstract heap H_{err}. This one has two extra nodes in addition to the nodes in H_0, and represents erroneous configurations in which the Label object is about to throw an exception in one of its methods. (We set a special error-bit whenever an exception is raised, and the node L_{err} represents an object where that bit is set.)

Technically (see Sections V and VI), nested graphs represent ideals of downward-closed sets (relative to graph embedding) of configurations of depth-bounded predicate abstractions of the heap. While the abstract state space is infinite, it has a good algebraic structure, and abstract reachability analysis can be shown to terminate [1], [9], [16].

Transitions: Object Maps Suppose we get a finite set $\mathcal{S}$ of abstract heaps represented as above. The second challenge is that method calls may have parameters and may change the state of the receiver object as well as objects reachable from it or even objects that can reach the receiver. As an example, consider a set container object with some iterators pointing to it. Removing an element through an iterator can change the state of the iterator (it may reach the end), the set (it can become empty), as well as other iterators associated with the set (they become invalidated and may not be used to traverse the set). Thus, transitions cannot simply be labeled with method names, but must also indicate which abstract objects participate in the call as well as the effect of the call on the abstract objects. The interface must describe the effect of the heap in all cases, and all methods. In our example, we can enumerate 14 possible transitions from H_0. To complete the description of an interface, we have to (1) show how a method call transforms the abstract heap, and (2) ensure that each possible method call from each abstract heap in $\mathcal{S}$ ends up in an abstract heap also in $\mathcal{S}$.

Consider invoking the set method of a viewer in the abstract heap H_0. There are several choices: one can choose V_d, V_{nd}, or V_0 as the receiver, and pass it L_d or L_{nd}. Note that the method call captures the scenario in which one representative object is chosen from each node and the method is executed. Recall that, because of stars, a single node may represent multiple objects. Figure 1(e) shows how the abstract heap is transformed if we choose V_{nd} as the receiver and pass it an L_d object. The following properties hold. First, both the source and the target of the transition are H_0, hence, the method call transforms the abstract heap H_0 back to H_0. We omit nodes of H_0 that are untouched by the transition. Second, updates to nodes are shown using directed hyperedges (dotted multi-destination lines in the figure). The hyperedges are optionally labeled with "1" (e.g., the hyperedge from L_{nd} to the destination L_d). The hyperedge labeled 1 indicates that one object from the source node transfers to the state in the destination; every other node in the source node moves (non-deterministically) to some destination of the hyperedge (that is not marked by "1"). So, the method call $V_{nd}.\mathtt{set}(L_d)$ moves one viewer object from V_{nd} to V_d (the callee), and all other viewer objects in state V_{nd} non-deterministically to either V_d or V_{nd}. At the same time, one L_{nd} object (the old viewer pointed to by the callee) moves to L_d, and all other remain in L_{nd}. Finally, all L_d objects (including the parameter of the call) remain in L_d. Intuitively, this captures the situation that the previous (non-disposed) label object is now disposed, and the viewer now points to a disposed label.

The second transition shows what happens if set is called on V_d with any label. This time, an error occurs, since the method call tries to dispose an already disposed label. This is indicated by a transition to the error node H_{err}, and thus, is not allowed in the interface.

Algorithm for Interface Computation Our second contribution is an algorithm and a tool for computing the dynamic package interfaces in form of a state machine, as described above. To compute the interface of a package, we fix a set of predicates and then perform an abstract reachability analysis of the package together with a *most general client*. Intuitively, the most general client [7] runs in an infinite loop; in each iteration of the loop, it non-deterministically either allocates a new object of some class, or picks an already allocated object, a public method of the object, a sequence of arguments to the method, and invokes the method call on the object. This way, it explores all possible sequences of constructors and method calls. The properties of ideal abstraction of depth-bounded systems [17] ensures that the abstract reachability analysis terminates and produces a finite coverability tree for the package. The nodes of the coverability tree represent abstract heaps and edges represent object maps on transitions. We take the maximal nodes of this tree (relative to embedding of abstract heaps) as the states of our interface, and the associated transitions as the transitions.

In our example, there are two maximal nodes: H_0 and H_{err}, where H_{err} denotes the error condition. Accordingly, the interface shows that H_0 captures the "most general" abstract heap in the use of this package, each "correct" method call preserves H_0, and also indicates the method calls that will lead to H_{err} and should not be allowed. We omit showing the remaining 12 transitions.

III. Preliminaries

A *quasi-ordering* $\leq$ is a reflexive and transitive relation $\leq$ on a set X. In the following $X(\leq)$ is a quasi-ordered set. The *downward closure* (resp. *upward closure*) of $Y \subseteq X$ is $\downarrow Y = \{ x \in X \mid \exists y \in Y.\, x \leq y \}$ (resp. $\uparrow Y = \{ x \in X \mid \exists y \in Y.\, y \leq x \}$). A set Y is *downward-closed* (resp. *upward-closed*) if $Y = \downarrow Y$ (resp. $Y = \uparrow Y$). An element $x \in X$ is an *upper bound* for $Y \subseteq X$ if for all $y \in Y$ we have $y \leq x$. A nonempty set $D \subseteq X$ is *directed* if any two elements in D have a common upper bound in D. A set $I \subseteq X$ is an *ideal* of X if I is downward-closed and directed. A quasi-ordering $\leq$ on a set X is a *well-quasi-ordering* (wqo) if any infinite sequence $x_0, x_1, x_2, \ldots$ of elements from X contains an increasing pair $x_i \leq x_j$ with $i < j$.

A *transition system* $\mathcal{S} = (X, X_0, \to)$ consists of a set X of states, a set $X_0 \subseteq X$ of initial states, and a transition relation $\to \subseteq X \times X$. We write $x \to x'$ for $(x, x') \in \to$. We define the *post operator* as $\mathsf{post}.\mathcal{S} : \mathcal{P}(X) \to \mathcal{P}(X)$ with $\mathsf{post}.\mathcal{S}(Y) = \{ x' \in X \mid \exists x \in Y.\, x \to x' \}$. The *reachability set* of a transition system $\mathcal{S}$, denoted $\mathit{Reach}(\mathcal{S})$, is defined by $\mathit{Reach}(\mathcal{S}) = \mathit{lfp}^{\subseteq}(\lambda Y. X_0 \cup \mathsf{post}.\mathcal{S}(Y))$. A *well-structured transition system* (WSTS) is a tuple $\mathcal{S} = (X, X_0, \to, \leq)$ where $(X, X_0, \to)$ is a transition system and $\leq \subseteq X \times X$ is a wqo that is *monotonic* with respect to $\to$, i.e., for all x_1, x_2, y_1, t such that $x_1 \leq y_1$ and $x_1 \to x_2$, there exists y_2 such that $y_1 \to y_2$ and $x_2 \leq y_2$. The *covering set* of a well-structured transition system $\mathcal{S}$, denoted $\mathit{Cover}(\mathcal{S})$, is defined by $\mathit{Cover}(\mathcal{S}) = \downarrow \mathit{Reach}(\mathcal{S})$.

IV. Concrete Semantics

We now present a core OO language and its semantics.

Syntax For a set of variables X, we denote by $\mathsf{Exp}.X$ and $\mathsf{Pred}.X$ the set of expressions and predicates respectively, with variables drawn from $\{\mathsf{this}.x \mid x \in X\}$. We assume there are two special variables this and null.

In our language, a package consists of a collection of class definitions. A class definition consists of a class name, a constructor method, a set of fields, and a set of method declarations partitioned into public and protected methods. A constructor method has the same name as the class, a list of typed arguments, and a body. We assume fields are typed with either a finite scalar type (e.g., Boolean), or a class name. The former are called *scalar* fields and the latter *reference* fields. Intuitively, reference fields refer to other objects on the heap. Methods consist of a signature and a body. The signature of a method is a typed list of its arguments and its return value. The body of a method is given by a control flow automaton over the fields of the class. Intuitively, any client can invoke public methods, but only other classes in the package can invoke protected ones.

A control flow automaton (CFA) over a set of variables X and a set of operations $\mathsf{Op}.X$ is a tuple $F = (X, Q, q_0, q_f, T)$, where Q is a finite set of *control states*, $q_0 \in Q$ (resp. $q_f \in Q$) is a designated initial state (resp. final state), and $T \subseteq Q \times \mathsf{Op}.X \times Q$ is a set of edges labeled with operations.

For our language, we define the set $\mathsf{Op}.X$ of *operations* over X to consist of: (i) *assignments* $\mathsf{this}.x := e$, where $x \in X$ and $e \in \mathsf{Exp}.X$; (ii) *assumptions*, $\mathsf{assume}(p)$, where $p \in \mathsf{Pred}.X$, (iii) *construction* $\mathsf{this}.x = \mathsf{new}(C(\overline{a}))$, where C is a class name and $\overline{a}$ is a sequence in $\mathsf{Exp}.X$, and (iv) *method calls* $\mathsf{this}.x := \mathsf{this}.y.m(\overline{a})$, where $x, y \in X$.

Formally, a class $C = (A, c, M_p, M_t)$, where A is the set of fields, c is the constructor, M_p is the set of public methods and M_t is the set of protected methods. We use C also for the name of the class. A package P is a set of classes.

We make the following assumptions. First, all field and method names are disjoint. Second, each class has an attribute ret used to return values from a method to its callers. Third, all CFAs are over disjoint control locations. Finally, a package is well-typed, in that assignments are type-compatible, called methods exist and are called with the right number and types of arguments, etc. Finally, for simplicity, we omit recursive method calls in the analysis.

A *client* I of a package P is a class with exactly one method main, such that (i) for each $x \in I.A$, we have the type of x is either a scalar or a class name from P, (ii) in all method calls $\mathsf{this}.x = \mathsf{this}.y.m(\overline{a})$, m is a public method of its class, and (iii) edges of main can have the additional *non-deterministic assignment* $\mathsf{havoc}(\mathsf{this}.x)$. An OO program is a pair (P, I) of a package P and a client I.

Concrete Semantics Fix an OO program $S = (P, I)$. It induces a labelled transition system $(Conf, U_0, \to)$, with configurations $Conf$, initial configurations U_0, and transition relation $\to$ as follows.

Let $\mathcal{O}$ be a countably infinite set of *object identifiers* (or simply objects) and let $class : \mathcal{O} \to P \cup \{I, nil\}$ be a function mapping each object identifier to its class. A *configuration* $u \in Conf$ is a tuple $(O, this, q, \nu, st)$, where $O \subseteq \mathcal{O}$ is a finite set of currently allocated *objects*, $this \in O$ is the *current object* (i.e., the receiver of the call to the method currently executed), q is the *current control state*, which specifies the control state of the CFA at which the next operation will be performed, ν is a sequence of triples of object, variable, and control location (the program stack), and st is a *store*, which maps an object and a field to a value in its domain. We require that O contains a unique *null* object *null* with $class(null) = nil$. We denote by $Conf$ the set of all configurations of S.

The set of *initial configurations* $U_0 \subseteq Conf$ is the set of configurations $u_0 = (\{null, o_I\}, this, \mathsf{main}.q_0, \varepsilon, st)$ such that (i) $class(o_I) = I$, (ii) the current object $this = o_I$, (iii) the value of all reference fields of all objects in the store is *null* and all scalar fields take some default value in their domain, and (iv) the control state is the initial state of the CFA of the main method of I and the stack is empty.

Given a store, we write $st(e)$ and $st(p)$ for the value of an expression e or predicate p evaluated in the store st, computed the usual way.

The transitions in $\to$ are as follows. A configuration $u = (O, this, q, \nu, st)$ moves to configuration $u' = (O', this', q', \nu', st')$ if there is an edge (q, op, q') in the CFA of q such that

- $op = \mathsf{this}.x := e$ and $O' = O$, $this' = this$, $\nu' = \nu$, and $st' = st[(this, x) \mapsto st(e)]$.
- $op = \mathsf{assume}(p)$ and $O' = O$, $this' = this$, $\nu' = \nu$, $st(p) = 1$, and $st' = st$.
- $op = \mathsf{this}.x := \mathsf{this}.y.m(\overline{a})$ and $O' = O$, $this' = this$, $\nu' = (this, x, q')\nu$, and $q' = m.q_0$, and the formal arguments of m are assigned values $st(\overline{a})$ in the store.
- $op = \mathsf{this}.x := \mathsf{new}(C(\overline{a}))$ and $O' = O \uplus \{o\}$ for a new object o with $class(o) = C$, $this' = o$, $\nu' = (this, x, q')\nu$, and $q' = c.q_0$ for the constructor c of C, and the formal arguments of c are assigned values $st(\overline{a})$ in the store.
- $op = \mathsf{havoc}(\mathsf{this}.x)$: $O' = O$, $this' = this$, and $st' = st[(this, x) \mapsto v]$, where v is some value chosen non-deterministically from the domain of x.

Finally, if q is the final node of a CFA and $\nu = (o, x, q)\nu'$, and $u = (O, this, q, \nu, st)$ moves to configuration $u' = (O, o, q, \nu', st')$, where $st' = st[o.x \mapsto st(this.\mathsf{ret})]$. If none of the rules apply, the program terminates.

To model error situations, we assume that each class has a field *err* which is initially 0 and set to 1 whenever an error is encountered (e.g., an assertion is violated). An error configuration is a configuration u in which there exists an object $o \in u.O$ such that $o.err = 1$. An OO program is *safe* if it does not reach any error configuration.

V. Depth-Bounded Abstract Semantics

We now present an abstract semantics for OO programs. Given an OO program S, our abstract semantics of S is a labelled transition system $S^\#_{\mathsf{h}} = (Conf^\#, U^\#_0, \to^\#_{\mathsf{h}})$ that is obtained by an abstract interpretation [4] of S. Typically, the system $S^\#_{\mathsf{h}}$ is still an infinite state system. However, the abstraction ensures that $S^\#_{\mathsf{h}}$ belongs to the class of *depth-bounded systems* [9]. Depth-bounded systems are well-structured transition systems that can be effectively analyzed [16], and this will enable us to compute the dynamic package interface.

Heap Predicate Abstraction We start with a heap predicate abstraction, following shape analysis [13], [11]. Let AP be a finite set of *unary abstraction predicates* from $\mathsf{Pred}.(\{x\} \cup \mathcal{C}.A)$ where x is a fresh variable different from this and null. For a configuration $u = (O, \cdot, st)$ and $o \in O$, we write $u \models p(o)$ iff $st[x \mapsto o](p) = 1$. Further, let AR be a subset of the reference fields in $\mathcal{C}.A$. We refer to AR as *binary abstraction predicates*. For an object $o \in \mathcal{O}$, we denote by $AR(o)$ the set $AR \cap class(o).A$.

The concrete domain D of our abstract interpretation is the powerset of configurations $D = \mathcal{P}(Conf)$, ordered by subset inclusion. The abstract domain $D^\#_\mathsf{h}$ is the powerset of *abstract configurations* $D^\#_\mathsf{h} = \mathcal{P}(Conf^\#)$, again ordered by subset inclusion. An abstract configuration $u^\# \in Conf^\#$ is like a concrete configuration except that the store is abstracted by a finite labelled graph, where nodes are object identifiers, edges correspond to the values of reference fields in AR, and node labels denote the evaluation of objects on the predicates in AP. That is, the abstract domain is parameterized by both AP and AR.

Formally, an abstract configuration $u^\# \in Conf^\#$ is a tuple $(O, this, q, \nu, \eta, st)$ where $O \subseteq \mathcal{O}$ is a finite set of object identifiers, $this \in O$ is the current object, $q \in F.Q$ is the current control location, ν is a finite sequence of triples (o, x, q) of objects, variables, and control location, $\eta : O \times AP \to \mathbb{B}$ is a *predicate valuation*, and st is an *abstract store* that maps objects in $o \in O$ and reference fields $a \in AR(o)$ to objects $st(p, a) \in \mathsf{Val}.a$. Note that we identify the elements of $Conf^\#$ up to isomorphic renaming of object identifiers.

The meaning of an abstract configuration is given by a concretization function $\gamma_\mathsf{h} : Conf^\# \to D$ defined as follows: for $u^\# \in Conf^\#$ we have $u \in \gamma_\mathsf{h}(u^\#)$ iff (i) $u^\#.O = u.O$; (ii) $u^\#.this = u.this$; (iii) $u^\#.q = u.q$; (iv) $u^\#.\nu = u.\nu$; (v) for all $o \in u.O$ and $p \in AP$, $u^\#.\eta(o, p) = 1$ iff $u \models p(o)$; and (vi) for all public objects $o \in O$, and $a \in AR(o)$, $u.st(o, a) = u^\#.st(o, a)$. We lift γ_h pointwise to a function $\gamma_\mathsf{h} : D^\#_\mathsf{h} \to D$ by defining $\gamma_\mathsf{h}(U^\#) = \bigcup\{\, \gamma_\mathsf{h}(u^\#) \mid u^\# \in U^\# \,\}$. Clearly, γ_h is monotone. It is also easy to see that γ_h distributes over meets because for each configuration u there is, up to isomorphism, a unique abstract configuration $u^\#$ such that $u \in \gamma_\mathsf{h}(u^\#)$. Hence, let $\alpha_\mathsf{h} : D \to D^\#_\mathsf{h}$ be the unique function such that $(\alpha_\mathsf{h}, \gamma_\mathsf{h})$ forms a Galois connection between D and $D^\#_\mathsf{h}$, i.e., $\alpha_\mathsf{h}(U) = \bigcap\{\, U^\# \mid U \subseteq \gamma_\mathsf{h}(U^\#) \,\}$.

The abstract transition system $S^\#_\mathsf{h} = (Conf^\#, U^\#_0, \to^\#_\mathsf{h})$ is obtained by setting $U^\#_0 = \alpha_\mathsf{h}(U_0)$ and defining $\to^\#_\mathsf{h} \subseteq Conf^\# \times Conf^\#$ as follows. Let $u^\#, v^\# \in Conf^\#$. We have $u^\# \to^\#_\mathsf{h} v^\#$ iff $v^\# \in \alpha_\mathsf{h} \circ \mathsf{post}.S.t \circ \gamma_\mathsf{h}(u^\#)$.

Theorem 5.1: The system $S^\#_\mathsf{h}$ simulates the concrete system S, i.e., (i) $U_0 \subseteq \gamma_\mathsf{h}(U^\#_0)$ and (ii) for all $u, v \in Conf$ and $u^\# \in Conf^\#$, if $u \in \gamma_\mathsf{h}(u^\#)$ and $u \to v$, then there exists $v^\# \in Conf^\#$ such that $u^\# \to^\#_\mathsf{h} v^\#$ and $v \in \gamma_\mathsf{h}(v^\#)$.

Depth-Boundedness Let $u^\# \in Conf^\#$ be an abstract configuration. A *simple path* of length n in $u^\#$ is a sequence of distinct objects $\pi = o_1, \ldots, o_n$ in $u^\#.O$ such that for all $1 \leq i < n$, there exists $a_i \in AR(o_i)$ with $u^\#.st(o_i, a_i) = a_{i+1}$. We denote by $lsp(u^\#)$ the length of the longest simple path of $u^\#$. We say that a set of abstract configurations $U^\# \subseteq Conf^\#$ is *depth-bounded* if $U^\#$ is bounded in the length of its simple paths, i.e., there exists $k \in \mathbb{N}$ such that $k = \sup_{u^\# \in U^\#} lsp(u^\#)$.

We next show that under certain restrictions on the binary abstraction predicates AR, the abstract transition system $S^\#_\mathsf{h}$ is a well-structured transition system. For this purpose, we define the *embedding order* on abstract configurations. An *embedding* for two configurations $u^\#, v^\# : Conf^\#$ is a function $h : u^\#.O \to v^\#.O$ such that the following conditions hold: (i) h preserves the class of objects: for all $o \in u^\#.O$, $class(o) = class(h(o))$; (ii) h preserves the current object, $h(u^\#.this) = v^\#.this$; (iii) h preserves the stack, $\bar{h}(u^\#.\nu) = v^\#.\nu$ where $\bar{h}$ is the unique extension of h to stacks; (iv) h preserves the predicate valuation: for all $o \in u^\#.O$ and $p \in AP$, $u^\#.\eta(o, p)$ iff $v^\#.\eta(h(o), p)$; and (v) h preserves the abstract store, i.e., for all $o \in u^\#.O$ and $a \in AR(o)$, we have $h(u^\#.st^\#(o, a)) = v^\#.st^\#(h(o), a)$. The embedding order $\preceq$: $Conf^\# \times Conf^\#$ is then as follows: for all $u^\#, v^\# : Conf^\#$, $u^\# \preceq v^\#$ iff $u^\#$ and $v^\#$ share the same current control location ($u^\#.q = v^\#.q$) and there exists an injective embedding of $u^\#$ into $v^\#$.

Lemma 5.2: (1) The embedding order is monotonic with respect to abstract transitions in $S^\#_\mathsf{h} = (Conf^\#, U^\#_0, \to^\#_\mathsf{h})$. (2) Let $U^\#$ be a depth-bounded set of abstract configurations. Then $(U^\#, \preceq)$ is a wqo.

Theorem 5.3: If $Reach(S^\#_\mathsf{h})$ is depth-bounded, then $(Reach(S^\#), U^\#_0, \to^\#_\mathsf{h}, \preceq)$ is a WSTS.

In practice, we can ensure depth-boundedness of $Reach(S^\#_\mathsf{h})$ syntactically by choosing the set of binary abstraction predicates AR such that it does not contain reference fields that span recursive data structures. Such reference fields are only allowed to be used in the defining formulas of the unary abstraction predicates. In the next section, we assume that the set $Reach(S^\#_\mathsf{h})$ is depth-bounded and we identify $S^\#_\mathsf{h}$ with its induced WSTS.

VI. Ideal Abstraction

The set of abstract error configurations is upward-closed with respect to the embedding order $\preceq$, i.e., we have $U^\#_{err} = {\uparrow} U^\#_{err}$. From the monotonicity of $\preceq$ we therefore conclude that $Reach(S^\#_\mathsf{h}) \cap U^\#_{err} = \emptyset$ iff $Cover(S^\#_\mathsf{h}) \cap U^\#_{err} = \emptyset$. This means that if we analyze the abstract transition system $S^\#_\mathsf{h}$ modulo downward closure of abstract configurations, this does not incur an additional loss of precision. We exploit this observation as well as the fact that $S^\#_\mathsf{h}$ is well-structured to construct a finite abstract transition system whose configurations are given by downward-closed sets of abstract configurations. We then show that this abstract transition system can be effectively computed.

The key insight is that every downward-closed subset of a well-quasi ordered set is a *finite* union of ideals. That is, if we can finitely represent ideals of abstract configurations, we can finitely represent arbitrary downward-closed sets. We formalize this observation in abstract interpretation, and refer to this abstraction as *ideal abstraction* [17].

The abstract domain $D^\#_{\mathsf{idl}}$ of the ideal abstraction is given by downward-closed sets of abstract configurations, which we represent as finite sets of ideals. The concrete domain is $D^\#_{\mathsf{h}}$. The ordering on the abstract domain is subset inclusion. The abstraction function is downward closure.

Formally, we denote by $Idl(Conf^\#)$ the set of all depth-bounded ideals of abstract configurations with respect to the embedding order. The abstract domain $D^\#_{\mathsf{idl}}$ is $\mathcal{P}_{\mathsf{fin}}(Idl(Conf^\#))$. The concretization function $\gamma_{\mathsf{idl}} : D^\#_{\mathsf{idl}} \to D^\#_{\mathsf{h}}$ is $\gamma_{\mathsf{idl}}(\mathcal{I}) = \bigcup \mathcal{I}$. Further, define the abstraction function $\alpha_{\mathsf{idl}} : D^\#_{\mathsf{h}} \to D^\#_{\mathsf{idl}}$ as $\alpha_{\mathsf{idl}}(U^\#) = \{ I \in Idl(Conf^\#) \mid I \subseteq \downarrow U^\# \}$. From the ideal abstraction framework [17], it follows that $(\alpha_{\mathsf{idl}}, \gamma_{\mathsf{idl}})$ forms a Galois connection between $D^\#_{\mathsf{h}}$ and $D^\#_{\mathsf{idl}}$. The overall abstraction is then given by the Galois connection (α, γ) between D and $D^\#_{\mathsf{idl}}$, which is defined by $\alpha = \alpha_{\mathsf{idl}} \circ \alpha_{\mathsf{h}}$ and $\gamma = \gamma_{\mathsf{h}} \circ \gamma_{\mathsf{idl}}$. We define the *abstract post operator* $\mathsf{post}^\#$ of S as the most precise abstraction of $\mathsf{post}.S$ with respect to this Galois connection, i.e., $\mathsf{post}^\#.S = \alpha \circ \mathsf{post}.S \circ \gamma$.

In the following, we assume the existence of a *sequence widening operator* $\nabla_{\mathsf{idl}} : Idl(Conf^\#)^+ \rightharpoonup Idl(Conf^\#)$, i.e., ∇_{idl} satisfies the following two conditions: (i) *covering condition*: for all $\mathcal{I} \in Idl(Conf^\#)^+$, if $\nabla_{\mathsf{idl}}(\mathcal{I})$ is defined, then for all I in $\mathcal{I}$, $I \subseteq \nabla_{\mathsf{idl}}(\mathcal{I})$.; and (ii) *termination condition*: for every ascending chain $(I_i)_{i\in\mathbb{N}}$ in $Idl(Conf^\#)$, the sequence $J_0 = I_0$, $J_i = \nabla_{\mathsf{idl}}(I_0 \dots I_i)$, for all $i > 0$, is well-defined and an ascending stabilizing chain. The actual definition of ∇_{idl} is provided in the extended version of this paper.

The ideal abstraction induces a finite labeled transition system $S^\#_{\mathsf{idl}}$ whose configurations are ideals of abstract configurations. There are special transitions labeled with ϵ, which we refer to as *covering transitions*. We call $S^\#_{\mathsf{idl}}$ the *abstract coverability DAG* of $S^\#_{\mathsf{h}}$. This is because the set of reachable configurations of $S^\#_{\mathsf{idl}}$ over-approximates the covering set of $S^\#_{\mathsf{h}}$, i.e., $Cover(S^\#_{\mathsf{h}}) \subseteq \gamma_{\mathsf{idl}}(Reach(S^\#_{\mathsf{idl}}))$. Furthermore, the directed graph spanned by the non-covering transitions of $S^\#_{\mathsf{idl}}$ is acyclic.

Formally, we define $S^\#_{\mathsf{idl}} = (\mathcal{I}_{\mathsf{idl}}, \mathcal{I}_0, \dot{\to}^\#_{\mathsf{idl}})$ as follows. The initial configurations $\mathcal{I}_0$ are given by $\mathcal{I}_0 = \alpha_{\mathsf{idl}}(U^\#_0)$. The set of configurations $\mathcal{I}_{\mathsf{idl}} \subseteq Idl(Conf^\#)$ and the transition relation $\dot{\to}^\#_{\mathsf{idl}} \subseteq \mathcal{I}_{\mathsf{idl}} \times \mathcal{I}_{\mathsf{idl}}$ are defined as the smallest sets satisfying the following conditions: (1) $\mathcal{I}_0 \subseteq \mathcal{I}_{\mathsf{idl}}$; and (2) for every $I \in \mathcal{I}_{\mathsf{idl}}$, let $paths(I)$ be the set of all sequences of ideals $I_0 \dots I_n$ with $n \geq 0$ such that $I_0 \in \mathcal{I}_0$, $I_n = I$, and for all $0 \leq i < n$, $I_i \dot{\to}^\#_{\mathsf{idl}} I_{i+1}$. Then, for every path $\mathcal{I} = I_0 \dots I_n \in paths(I)$, if there exists $i < n$ such that $I \subseteq I_i$, then $I \xrightarrow{\epsilon}^\#_{\mathsf{idl}} I_i$. Otherwise, for all $I' \in \mathsf{post}^\#.S \circ \gamma_{\mathsf{idl}}(I)$, let $J' = \nabla_{\mathsf{idl}}(\mathcal{I}'I')$ where $\mathcal{I}'$ is the subsequence of all ideals I_i in $\mathcal{I}$ with $I_i \subseteq I'$, then $J' \in \mathcal{I}_{\mathsf{idl}}$ and $I \dot{\to}^\#_{\mathsf{idl}} J'$.

Theorem 6.1: The abstract coverability DAG $S^\#_{\mathsf{idl}}$ is finite.

Define the relation $\xrightarrow{*}^\#_{\mathsf{idl}} \subseteq \mathcal{I}_{\mathsf{idl}} \times \mathcal{I}_{\mathsf{idl}}$ as $\xrightarrow{*}^\#_{\mathsf{idl}} = \dot{\to}^\#_{\mathsf{idl}} \cup \xrightarrow{\epsilon}^\#_{\mathsf{idl}} \circ \dot{\to}^\#_{\mathsf{idl}}$. We now state our main soundness theorem.

Theorem 6.2: **[Soundness]** The abstract coverability DAG $S^\#_{\mathsf{idl}}$ simulates S, i.e., (i) $U_0 \subseteq \gamma(\mathcal{I}_0)$ and (ii) for all $I \in \mathcal{I}_{\mathsf{idl}}$ and $u, v \in Reach(S)$, if $u \in \gamma(I)$ and $u \to v$, then there exists $J \in \mathcal{I}_{\mathsf{idl}}$ such that $v \in \gamma(J)$ and $I \xrightarrow{*}^\#_{\mathsf{idl}} J$.

In the rest of this section we explain how we represent ideals of abstract configurations and how the operations for computing the abstract coverability DAG are implemented.

Representing Ideals of Abstract Configurations: The ideals of depth-bounded abstract configurations are recognizable by regular hedge automata [16]. We can encode these automata into abstract configurations $I^\#$ that are equipped with a *nesting level function*. The nesting level function indicates how the substructures of the abstract store of $I^\#$ can be replicated to obtain all abstract configurations in the represented ideal.

Formally, a *quasi-ideal configuration* $I^\#$ is a tuple $(O, this, q, v, \eta, st, nl)$ where $nl : O \to \mathbb{N}$ is the nesting level function and $(O, this, q, v, \eta, st)$ is an abstract configuration, except that η is only a partial function $\eta : O \times AP \rightharpoonup \mathbb{B}$. We denote by $QIdlConf^\#$ the set of all quasi-ideal configurations. We call a quasi-ideal configuration $I^\# = (O, this, q, v, \eta, st, nl)$ simply *ideal configuration*, if η is total and for all $o \in O$, $a \in AR(o)$, $nl(o) \geq nl(st(o, a))$. We denote by $[I^\#]$ the inherent abstract configuration $(O, this, q, v, \eta, st)$ of an ideal configuration $I^\#$. Further, we denote by $IdlConf^\#$ the set of all ideal configurations and by $IdlConf^\#_0$ the set of all ideal configurations in which all objects have nesting level 0. We call the latter *finitary* ideal configurations.

Meaning of Quasi-Ideal Configurations: Let $I^\# = (O, this, q, v, st, nl)$ and $J^\# = (O', this', q', v', st', nl')$ be quasi-ideal configurations. An *inclusion mapping* between $I^\#$ and $J^\#$ is an embedding $h : O \to O'$ that satisfies the following additional conditions: (i) for all $o \in O$, $nl(o) \leq nl'(h(o))$; (ii) h is injective with respect to level 0 vertices in O': for all $o_1, o_2 \in O$, $o' \in O'$, $h(o_1) = h(o_2) = o'$ and $nl'(o') = 0$ implies $o_1 = o_2$; and (iii) for all distinct $o_1, o_2, o_3 \in O$, if $h(o_1) = h(o_2)$, and o_1 and o_2 are both neighbors of o_3, i.e., $st(o_3, a_1) = o_1$ and $st(o_3, a_2) = o_2$ for some $a_1, a_2 \in AR(o_3)$, then $nl(o_1) > nl(o_3)$ and $nl(o_2) > nl(o_3)$.

We write $I^\# \leq_h J^\#$ if $q = q'$, and h is an inclusion mapping between $I^\#$ and $J^\#$. We say that $I^\#$ is *included* in $J^\#$, written $I^\# \leq J^\#$, if $I^\# \leq_h J^\#$ for some h.

We define the meaning $[\![I^\#]\!]$ of a quasi-ideal configuration $I^\#$ as the set of all inherent abstract configurations of the finitary ideal configurations included in $I^\#$:

$$[\![I^\#]\!] = \{ [J^\#] \mid J^\# \in IdlConf^\#_0 \wedge J^\# \leq I^\# \}$$

We extend this function to sets of quasi-ideal configurations, as expected.

Proposition 6.3: Ideal configurations exactly represent the depth-bounded ideals of abstract configurations, i.e., $\{ [\![I^\#]\!] \mid I^\# \in IdlConf^\# \} = Idl(Conf^\#)$.

Since the relation $\leq$ is transitive, we also get:

Proposition 6.4: For all $I^\#, J^\# \in QIdlConf^\#$, $I^\# \leq J^\#$ iff $[\![I^\#]\!] \subseteq [\![J^\#]\!]$.

It follows that inclusion of (quasi-)ideal configurations can be decided by checking for the existence of inclusion mappings, which is an NP-complete problem.

Quasi-ideal configurations are useful as an intermediate representation of the images of the abstract post operator. They can be thought of as a more compact representation of sets of ideal configurations. In fact, any quasi-ideal configuration can be reduced to an equivalent finite set of ideal configuration. We denote the function performing this reduction by $reduce : QIdlConf^\# \to \mathcal{P}_{\text{fin}}(IdlConf^\#)$ and we extend it to sets of quasi-ideal configurations, as expected.

Computing the Abstract Post Operator: We next define an operator $\mathsf{Post}^\#.S$ that implements the abstract post operator $\mathsf{post}^\#.S$ on ideal configurations. In the following, we fix an ideal configuration $I^\# = (O, this, q, \nu, st, nl)$ and a transition $t = (q, op, q')$ in S. For transitions not enabled at $I^\#$, we set $\mathsf{Post}^\#.S.t(I^\#) = \emptyset$.

We reduce the computation of abstract transitions $[I^\#] \to u^\#$ to reasoning about logical formulas. For efficiency reasons, we implicitly use an additional Cartesian abstraction [3] in the abstract post computation that reduces the number of required theorem prover calls. For a set of variables X, we assume a *symbolic weakest precondition* operator $\mathsf{wp} : \mathsf{Op}.(C.A) \times \mathsf{Pred}.(X \cup C.A) \to \mathsf{Pred}.(X \cup C.A)$ that is defined as usual. In addition, we need a symbolic encoding of abstract configurations into logical formulas. For this purpose, define a function $\Gamma : O \to \mathsf{Pred}.(O \cup C.A)$ as follows: given $o \in O$, let $O(o)$ be the subset of objects in O that are transitively reachable from o in the abstract store st, then $\Gamma(o)$ is the formula

$$\Gamma(o) = \text{distinct}(O(o) \cup O(this)) \wedge \mathsf{this} = this \wedge \mathsf{null} = null \wedge \bigwedge_{o' \in O(o) \cup O(this)} \left(\bigwedge_{p \in AP} \eta(o', p) \cdot p(o') \wedge \bigwedge_{a \in AR(o')} o'.a = st(o'.a) \right)$$

$$\text{where } \eta(o', p) \cdot p(o') = \begin{cases} p(o') & \text{if } \eta(o', p) = 1 \\ \neg p(o') & \text{if } \eta(o', p) = 0. \end{cases}$$

Now, let $\mathcal{J}^\#$ be the set of all quasi-ideal configurations $J^\# = (O, this, q', \nu, \eta', st', nl)$ that satisfy the following conditions:

- $\Gamma(this) \wedge q$ is satisfiable, if $op = \text{assume}(q)$;
- for all $o \in O$, $p \in AP$, if $\Gamma(o) \models \mathsf{wp}(op, p(o))$, then $\eta'(o, p) = 1$, else if $\Gamma(o) \models \mathsf{wp}(op, \neg p(o))$, then $\eta'(o, p) = 0$, else $\eta'(o, p)$ is undefined;
- for all $o, o' \in O$, $a \in AR(o)$, if $\Gamma(o) \wedge \Gamma(o') \models \mathsf{wp}(op, o.a = o')$, then $st'(o, a) = o'$, else if $\Gamma(o) \wedge \Gamma(o') \models \mathsf{wp}(op, o.a \neq o')$, then $st'(o, a) \neq o'$.

Then define $\mathsf{Post}^\#.S.t(I^\#) = reduce(\mathcal{J}^\#)$.

VII. Computing the Dynamic Package Interface

We now describe how to compute the dynamic package interface for a given package P. The computation proceeds in three steps. First, we compute the OO program $S = (P, I)$ that is obtained by extending P with its universal client I. Next, we compute the abstract coverability DAG $S^\#_{\mathsf{idl}}$ of S as described in Sections V and VI. We assume that the user provides sets of unary and binary abstraction predicates AP, respectively, AR that define the heap abstraction. Alternatively, we can use heuristics to guess these predicates from the program text of the package. Finally, we extract the package interface from the computed abstract coverability DAG. We describe this last step in more detail.

We can interpret the abstract coverability DAG as a numerical program. The control locations of this program are the ideal configurations in $S^\#_{\mathsf{idl}}$. With each abstract object occuring in an ideal configuration we associate a counter. The value of each counter denotes the number of concrete objects represented by the associated abstract object. While computing $S^\#_{\mathsf{idl}}$, we do some extra book keeping and compute for each transition of $S^\#_{\mathsf{idl}}$ a corresponding numerical transition that updates the counters of the counter program. These updates capture how many concrete objects change their representation from one abstract object to another.

The dynamic package interface $DPI(P)$ of P is a numerical program that is an abstraction of the numerical program associated with $S^\#_{\mathsf{idl}}$. The control locations of $DPI(P)$ are the ideal configurations in $S^\#_{\mathsf{idl}}$ that correspond to call sites, respectively, return sites to public methods of classes in P, in the universal client. A connecting path in $S^\#_{\mathsf{idl}}$ for a pair of such call and return sites (along with all covering transitions connecting ideal configurations on the path) corresponds to the abstract execution of a single method call. We refer to the restriction of the numerical program $S^\#_{\mathsf{idl}}$ to such a path and all its covering transitions as a *call program*. Each transition of $DPI(P)$ represents a summary of one such call program. Hence, a transition of $DPI(P)$ describes, both, how a method call affects the state of objects in a concrete heap configuration and how many objects are effected.

Note that a call program may contain loops because of loops in the method executed by the call program. The summarization of a call program therefore requires an additional abstract interpretation. The concrete domain of this abstract interpretation is given by transitions of counter programs, i.e., relations between valuations of counters. The concrete fixed point is the transitive closure of the transitions of the call program. The abstract domain provides an appropriate abstraction of numerical transitions. How precisely the package interface captures the possible sequences of method calls depends on the choice of this abstract domain and how convergence of the analysis of the call programs is enforced. We chose a simple abstract domain of *object mappings* that distinguishes between a constant number, respectively, arbitrary many objects transitioning from an abstract object on the call site of a method to another on the return site. However, other choices are feasible for this abstract domain that provide more or less information than object mappings.

VIII. Experiences

We have implemented our system by extending the Picasso tool [17].[1] Picasso uses an ideal abstraction to compute abstract coverability DAGs of depth-bounded graph rewriting systems. Our extension of Picasso computes a dynamic package interface from a graph rewriting system that encodes the semantics of the method calls in a package. In addition to the Viewer and Label example, described in Section II, we have experimented with other examples: a set and iterator, and JDBC statements and results.

Set and Iterator We considered a simple implementation of the Set and Iterator classes in which the items in a set are stored in a linked list. The Iterator class has the usual next, has_next, and remove methods. The Set class provides a method iterator, which creates an Iterator object associated with the set, and an add method, which adds a data element to the set. The interface of the package is meant to avoid raising exceptions of types NoSuchElementException and ConcurrentModificationException. A NoSuchElementException is raised whenever the next method is called on an iterator of an empty list. A ConcurrentModificationException is raised whenever an iterator accesses the set after the set has been modified, either through a call to the add method of the set or through a call to the remove method of another iterator. An iterator that removes an element can still safely access the set afterwards. (Similar restrictions apply to other Collection classes that implement Iterable.)

To obtain a depth-bounded abstraction of the package, without compromising soundness, we excluded the reference attributes of the class implementing the nodes of the linked list from the set of binary abstraction predicates. The unary abstraction predicate $empty(s)$ determines whether the size of a Set object s is zero or not. For Iterator objects, we specified two predicates that rely on the attributes of both the Set and the Iterator classes. The predicate $sync(i)$ holds for an Iterator object i that has the same version as its associated Set object. The predicate $mover(i)$ specifies that the position of an Iterator object i in the list of its associated Set object is less than the size of the set.

Our algorithm computes two maximal configurations H_0 and H_1. Figure 2 shows H_1; H_0 is the same as H_1, but with Set labeled with *empty*. There are also eight error abstract heap configurations, which correspond to different cases in which one of the two exceptions is raised for an Iterator object. Figure 3(a) and 3(b) show the object mappings of two transitions. For the sake of clarity, we have omitted the name of the reference attribute *iter_of* in the mappings. While both transitions invoke the remove() method on an Iterator object whose *mover* and *sync* predicates are true, they have different effects because they capture different concrete heaps represented by the same abstract heap H_1.

[1]Our tool and the full results of our experiments can be found at: http://pub.ist.ac.at/~zufferey/picasso/dpi/index.html

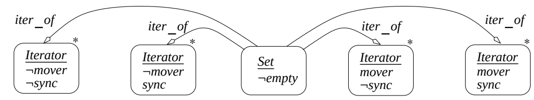
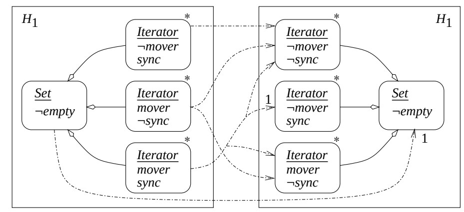
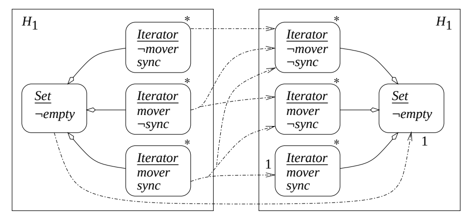


Figure 2. Abstract heap configuration H_1 of the set-iterator package using predicates: $empty(s) \equiv s.\mathtt{size} = 0$, $synch(i) \equiv i.\mathtt{iver} = i.\mathtt{iter\_of}.\mathtt{sver}$, and $mover(i) \equiv i.\mathtt{pos} < i.\mathtt{iter\_of}.\mathtt{size}$.



(a) Object mapping for $\underline{Iterator}(mover, sync)$.remove



(b) Object mapping for $\underline{Iterator}(mover, sync)$.remove

Figure 3. Two of the object mappings of the set-iterator interface

The first transition shows the case when the callee object becomes a non-mover; i.e., before the call to remove, its pos field refers to the last element of the linked list. The second shows the case when the callee object remains a mover, i.e., its pos field does not refer to the last element of the list. In both transitions, the other Iterator objects that reference the same Set object all become unsynced. Some of these objects remain movers while some of them become non-movers. In both cases, the callee remains sycned. There are two other symmetric transitions that capture the cases in which the Set object becomes empty. For the sake of brevity, we have presented the interface for a package which only allows a single Set object, however, the interface for the case when there is more than one Set object is similar, except that each abstract heap has an extra level of nesting.

JDBC (Java Database Connectivity) is a Java technology that enables access to databases of different types. We looked at three classes of JDBC for simple query access to databases: Connection, Statement, and ResultSet. A Connection object provides a means to connect to a database. A Statement object can execute an SQL query statement through a Connection object. A ResultSet object stores the result of the execution of a Statement object. All objects can be closed explicitly. If a Statement object is closed, its corresponding ResultSet object is also implicitly closed. Similarly, if a Connection object is closed, its corresponding Statement objects are implicitly closed, and so are the open ResultSet objects of these Statement objects. Java docu-

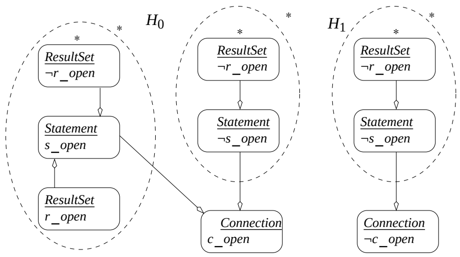


Figure 4. Abstract heap configurations H_0 and H_1 of JDBC package

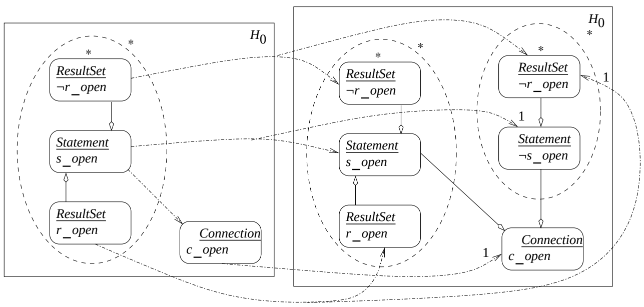


Figure 5. Object mapping for *Statement*(s_open).close()

mentation states: "By default, only one ResultSet object per Statement object can be open at the same time. Therefore, if the reading of one ResultSet object is interleaved with the reading of another, each must have been generated by different Statement objects. All execution methods in the Statement interface implicitly close a statement's current ResultSet object if an open one exists."

Figure 4 shows the two non-error maximal heaps H_0 and H_1 computed by our tool. These represent all safe configurations in which the Connection object is either open or closed. (For clarity, we have presented the case in which there is at most one Connection object by removing one nesting level.) Each type of object has a corresponding "open" predicate that specifies whether it is open or not. We omit showing abstract heaps capturing erroneous configurations. Lastly, Figure 5 shows the object mapping for the invocation of the close method on an open Statement object with an open ResultSet object. The mapping takes the Statement object and the open ResultSet object to a closed Statement and a closed ResultSet object. All other objects remain the same.

IX. Conclusions

We have formalized DPIs for OO packages with inter-object references, developed a novel ideal abstraction for heaps, and given a sound and terminating algorithm to compute DPIs on the (infinite) abstract domain. In contrast to previous techniques for multiple objects based on mixed static-dynamic analysis [10], [12], our algorithm is guaranteed to be sound. While our algorithm is purely static, an interesting future direction is to effectively combine it with dual, dynamic [5], [12] and template-based [14] techniques.

References

[1] P. A. Abdulla, K. Čerāns, B. Jonsson, and Yih-Kuan Tsay. General decidability theorems for infinite-state systems. In *LICS 96*, pages 313–321. IEEE, 1996.

[2] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *POPL'05*, pages 98–109. ACM, 2005.

[3] T. Ball, A. Podelski, and S.K. Rajamani. Boolean and cartesian abstraction for model checking C programs. *STTT*, 5(1):49–58, 2003.

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

[5] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. In *ISSTA*, pages 85–96. ACM, 2010.

[6] D. Giannakopoulou and C.S. Pasareanu. Interface generation and compositional verification in JavaPathfinder. In *FASE*, volume 5503 of *LNCS*, pages 94–108. Springer, 2009.

[7] T.A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In Michel Wermelinger and Harald Gall, editors, *ESEC/SIGSOFT FSE*, pages 31–40. ACM, 2005.

[8] Z. Li and Y.Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE-13*, pages 306–315. ACM, 2005.

[9] R. Meyer. On boundedness in depth in the pi-calculus. In *TCS*, IFIP 273, pages 477–489. Springer, 2008.

[10] M.G. Nanda, C. Grothoff, and S. Chandra. Deriving object typestates in the presence of inter-object references. In *OOPSLA*, pages 77–96. ACM, 2005.

[11] A. Podelski and T. Wies. Boolean heaps. In *SAS*, volume 3672 of *LNCS*, pages 268–283. Springer, 2005.

[12] M. Pradel, C. Jaspan, J. Aldrich, and T.R. Gross. Statically checking API protocol conformance with mined multi-object specifications. In *ICSE'12*, pages 925–935. IEEE, 2012.

[13] M. Sagiv, T.W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 24(3):217–298, 2002.

[14] A. Wasylkowski and A. Zeller. Mining temporal specifications from object usage. *Autom. Softw. Eng.*, 18(3-4):263–292, 2011.

[15] J. Whaley, M.C. Martin, and M.S. Lam. Automatic extraction of object-oriented component interfaces. In *ISSTA*, pages 218–228, 2002.

[16] T. Wies, D. Zufferey, and T.A. Henzinger. Forward analysis of depth-bounded processes. In *FOSSACS*, volume 6014 of *LNCS*, pages 94–108. Springer, 2010.

[17] D. Zufferey, T. Wies, and T.A. Henzinger. Ideal abstractions for well-structured transition systems. In *VMCAI*, volume 7148 of *LNCS*, pages 445–460. Springer, 2012.