Dynamic Package Interfaces

Shahram Esmaeilsabzali[1]⋆, Rupak Majumdar[2], Thomas Wies[3], and Damien Zufferey[4]⋆⋆

[1]University of Waterloo [2]MPI-SWS [3]NYU [4]MIT CSAIL

Abstract. A hallmark of object-oriented programming is the ability to perform computation through a set of interacting objects. A common manifestation of this style is the notion of a *package*, which groups a set of commonly used classes together. A challenge in using a package is to ensure that a client follows the implicit protocol of the package when calling its methods. Violations of the protocol can cause a runtime error or latent invariant violations. These protocols can extend across different, potentially unboundedly many, objects, and are specified informally in the documentation. As a result, ensuring that a client does not violate the protocol is hard.

We introduce *dynamic package interfaces (DPI)*, a formalism to explicitly capture the protocol of a package. The DPI of a package is a finite set of rules that together specify how any set of interacting objects of the package can evolve through method calls and under what conditions an error can happen. We have developed a dynamic tool that automatically computes an approximation of the DPI of a package, given a set of abstraction predicates. A key property of DPI is that the unbounded number of configurations of objects of a package are summarized finitely in an abstract domain. This uses the observation that many packages behave monotonically: the semantics of a method call over a configuration does not essentially change if more objects are added to the configuration. We have exploited monotonicity and have devised heuristics to obtain succinct yet general DPIs. We have used our tool to compute DPIs for several commonly used Java packages with complex protocols, such as JDBC, HashSet, and ArrayList.

1 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 and C# provide a clear specification of the static APIs of a package in terms of classes and their (typed) methods, there is usually no specification of the implicit *protocol* that constrains the temporal ordering of method calls on different objects. If the protocol is limited to a single object of a single class, it can be specified in form of a state machine whose states are the abstract states of the object and whose edges are the invocations of its methods [2, 14, 16]. For example, a lock object has two states: locked and unlocked. While in the unlocked (resp. locked) state, a call to the lock (resp. unlock) method takes

⋆ Shahram Esmaeilsabzali was at MPI-SWS when this work was done.

⋆⋆ Damien Zufferey was at IST Austria when this work was done.

it to the locked (resp. unlocked) state. Any other method call results in an error. The notion of state-machine interfaces has been studied extensively, and there are many tools to generate interfaces using static or dynamic techniques [2, 9, 13, 15]. However, existing notions of state machines on object states must be generalized when considering a package. First, the internal state of an object should be considered in the context of the internal states of other objects; e.g., in the Java Database Connectivity (JDBC) package, a `Statement` object can execute safely only if its corresponding `Connection` object is open. Second, the execution of a method on an object can change the internal state of other objects in the environment; e.g., calling the `executeQuery` method on a JDBC `Statement` object closes its corresponding open `ResultSet` object. Finally, the protocol can constrain the states and transitions of *unboundedly* many interacting objects; e.g., considering a collection object and its iterators, modifying the collection directly invalidates *all* of its iterators.

The problem of generalizing interfaces from single to multiple objects has been studied recently [10–12]. However, what is missing is a clear definition of what constitutes an interface in the presence of unboundedly many objects on the heap. Our first contribution is the introduction of *dynamic package interface* (DPI), which allows to capture the protocol of a package in a succinct manner. The DPI of a package is a set of *rules*, each of which specifies the effect of a method call on an object within an abstract *configuration* of objects. An abstract configuration denotes an unbounded number of concrete configurations of objects from a package. A rule has a *source* and a *destination* configuration, together with a *mapping* that specifies how the objects in the source change to the objects in the destination.

Our first technical ingredient is a representation of abstract configurations using *nested graphs* [17]. In a nested graph, a subgraph can be marked to be repeatable, and repetitions can be nested. Nested graphs naturally represent unbounded heap configurations. For example, Figure 1 shows a (two-level) nested graph representing an open JDBC `Connection` object with its many corresponding closed `Statement` objects, each with many closed `ResultSet` objects.

Our second ingredient is an abstract semantics of Java-like languages over the domain of nested graphs that is monotonic (in fact, the abstract transition system is *well-structured* [1]): if a method can be called in a "smaller" configuration, it can be also called in a "larger" configuration, with the resulting configurations maintaining the relationship. Monotonicity enables us to define the DPI rules of a package only over its *maximal* abstract configurations, letting each rule subsume infinitely many similar "smaller" rules. We prove that the set of maximal configurations has a finite representation, and thus the DPI of a package has a finite number of rules [6].

Our second contribution is a dynamic analysis technique to compute an approximation of the DPI of a package directly from the source code. Our tool explores the usage scenarios of a package by running a *universal client* that in each of its finite number of steps, nondeterministically, either creates a new object or invokes a method of an existing object. Each step of the universal client results in a rule. The universal client can end up computing hundreds or thousands of distinct rules, which makes the resulting DPI practically not useful. The challenge is to generalize these rules to obtain a compact DPI by exploiting similarity. Often, a pair of rules for the same method

are incomparable only because their sources and destinations are slightly different. For example, in one rule for the `close` method of the `Statement` class, the source configuration has closed `ResultSet` objects but not an open one, and vice versa, another rule might have an open `ResultSet` object but not closed ones. It makes sense, however, to combine these two rules because the effect of the two rules are essentially the same: the `Statement` object and its open `ResultSet` object are closed.

We have devised three heuristics that generalize a set of explored rules into a smaller, more general set. Our *extrapolation* heuristic compares the configurations of different rules and deduces whether the configuration of a certain rule can be expanded by repeating part of it based on the repetitions observed in the configurations of other rules. Our *merge* heuristic combines two rules that are based on similar method invocations into one rule. Our *exception isolation* heuristic combines two similar exception rules into one. While merging is similar to the union of the two rules, exception isolation is closer to an intersection that isolates the root cause of an exception. Our heuristics are all grounded in the monotonicity property of our abstract semantics.

We have used our tool to compute the DPIs of Java packages such as JDBC (26 rules), HashSet (16 rules), and ArrayList (15 rules). The rules of these DPIs can be traced to their documentation, as well as to the programming errors discussed in online discussion groups. Our tool more often than not computes the expected number of rules for these packages, but not all these rules are the most general ones. Our tool never computes a rule that is not consistent with the behaviour of a package. This is an indication that our heuristics are effective.

A more formal treatment of our work can be found in the technical reports [5, 6].

2 Overview and Outline

We now explain the notion of DPI, and describe the main steps that our tool carries out to compute the DPI of a package. We use Java Database Connectivity (JDBC), a package that provides database connectivity, as our running example.

We consider four commonly-used classes of JDBC and their methods. The `DriverManager` class allows to create a new connection to a database by invoking its static `getConnection` method. The string parameter of the method specifies the type of database, its address, and the needed credentials to access it. A `Connection` object can serve multiple `Statement` objects, each of which can be used to read or change the content of the database. The `createStatement` method of the `Connection` class creates a new `Statement` object. SQL commands and queries are executed through the `execute` and `executeQuery` methods of the `Statement` class. Both methods accept a string argument that is an SQL statement. The `executeQuery` method returns a new `ResultSet` object, which is a collection of rows retrieved from the database; the `next` method can be used to traverse these rows. A `Connection`, `Statement`, or `ResultSet` object is *open* initially, but can be closed via their corresponding `close` methods. Invoking the `executeQuery` method on a `Statement` object causes an open `ResultSet` object that references it to be closed, while creating a new open `ResultSet` object. If an object, or one of the objects that it references directly or transitively, is closed, invoking a non-`close` method on it would raise an exception.

2.1 System Input

Besides the names of classes and the signatures of their methods, our tool receives a set of abstraction predicates over the attributes of the classes. A predicate is either *scalar*, defined over the simple, non-reference attributes of the classes, or *reference*, determining which objects of a class are related to which objects of another class via a certain reference attribute. For simplicity, we assume these predicates are input by the user, but standard techniques based on Boolean methods and reference-valued fields in classes can be used to identify these predicates [15].

For example, in JDBC, the `Statement` class has an `active` attribute that determines whether it is open or not. This attribute is a unary scalar predicate, but in general a scalar predicate may read multiple fields from referenced objects. We also use the `applicationConnection` field of the `Statement` class to define a reference predicate that determines which `Statement` object points to which `Connection` object. We define similar scalar predicates for the `Connection` and `ResultSet` classes, which determine whether their objects are open or closed. We also define a reference predicate that determines which `ResultSet` objects reference which `Statement` objects.

We require that the set of reference attributes do not create a cycle when evaluated over objects: i.e., when objects are considered as nodes and the true valuations of reference attributes as directed edges, the resulting graph is acyclic. This is necessary as some of our algorithms rely on computing the topological ordering of heap-related graphs. This requirement can be relaxed: it is possible to allow the more general class of the depth-bounded graphs [6].

2.2 Nested Object Graphs

The enabling technique that allows us to compute a succinct, general DPI for a package is the ability to model a *heap configuration*, i.e., a set of concrete (e.g., Java) objects in the heap that reference each other, as a *nested object graph*.

A nested object graph is a labeled, directed graph whose subgraphs can be marked as repeatable. The nodes of a nested object graph represent objects and its directed edges represent references between the objects. The nodes and edges of the graph are labelled according to the input scalar and reference abstraction predicates, respectively. When a subgraph of a nested object graph is marked as *repeatable*, it denotes that arbitrary-many sets of objects similar to the objects in the subgraph can exist in the heap. Repetition can be nested, and hence the name "nested object graph." As an example, the nested object graph in Figure 1 represents all possible heap configurations consisting of an open `Connection` object with zero or more (in fact, possibly unboundedly many) closed `Statement` objects, each of which has zero or more closed `ResultSet` objects. Repetitions are specified via "*" next to nodes or subgraphs. Node *C*, for example, which represents the `ResultSet` objects, is marked repeatable in a nested manner: each group of repeatable `ResultSet` objects is associated with a `Statement` object,

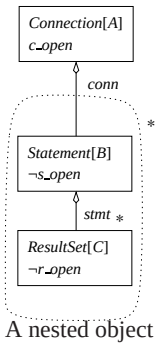


Fig. 1. A nested object graph

which itself is marked as repeatable via the "*" next to the subgraph specified by the dotted line. The repetition structure of a nested object graph is captured by assigning *nesting levels* to the nodes of the graph. The larger the nesting level is, the more levels of repetition it belongs to [6]. For example, the nesting levels of nodes *A*, *B*, and *C* in Figure 1 are 0, 1, and 2, respectively.

2.3 DPI Rules

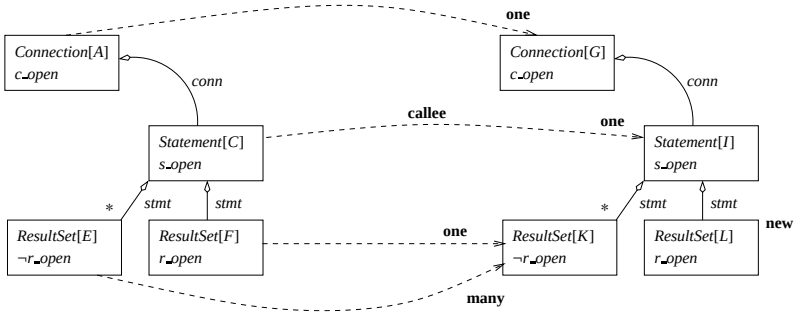
The *dynamic package interface* (DPI) of a package is a set of *rules*, each of which represents a family of method calls. A *rule* for a method call essentially specifies how a certain family of similar method calls change the shape of their corresponding heaps. A rule consists of:

- A *source* and a *destination* nested object graph, which represent all possible concrete heap configurations before and after the method call;
- A *source* and a *destination cast nested object graph*, each of which is a nested object graph some of whose nodes are labelled with *roles*, such as "callee", "parameter_0", and "new"; these graphs represent the heap configurations that are directly, in the sense that we will make clear, involved in the method call;
- An *object mapping*, which maps the nodes of the source nested object graph to the nodes of the destination nested object graph, possibly non-deterministically; and
- A *role mapping*, which maps the nodes of the source cast nested object graph to the nodes of the destination cast nested object graph; a node that is labelled by a role is mapped deterministically, but other nodes could be mapped non-deterministically.

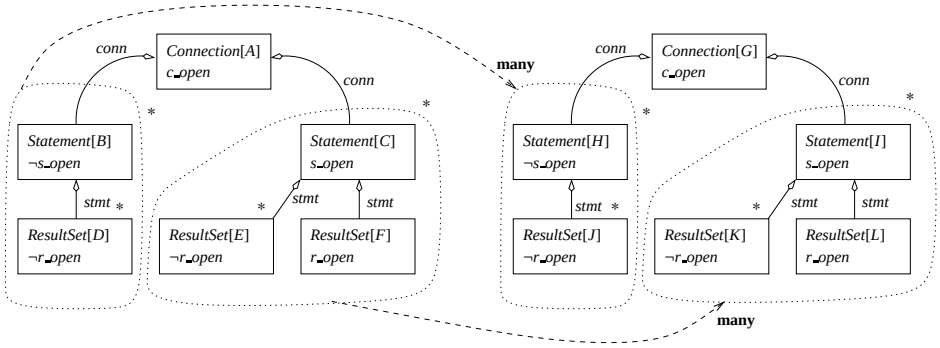
Each tuple in the object mapping or the role mapping is annotated with *multiplicity* information that specifies how many of the concrete objects represented by the source node are transferred to the destination node: *one* or *many*. The semantics of the computation of object mapping and role mapping of a rule should ensure that a concrete object is either mapped via the role mapping or the object mapping, but not both.

As an example, Figure 2 shows the rule that our system computes for `executeQuery` method calls that raise no exceptions. The rule specifies that an open `ResultSet` is closed when its corresponding `Statement` object performs `executeQuery`; instead, a new `ResultSet` object is created. Figure 2(a) specifies the role mapping of the rule, via dotted arrows that connect the nodes in the source cast nested object graph to the nodes in the destination cast nested object graph. The "callee" and "new" labels determine the callee and the newly created objects, respectively. Figure 2(b) specifies the object mapping of the rule via dotted arrows that, for the sake of brevity, connect the subgraphs of the nested object graphs. While in this rule the object mapping does not specify any change in its corresponding objects, in general that is not the case. Both nested object graphs and cast nested object graphs of the rule exhibit repetitions. It is this ability to express unbounded number of concrete heap configurations that allows us to compute general, yet concise rules.

Exception Rules. When a method call does not raise any exception, we are looking for general rules with the largest possible nested object graphs (because it captures more concrete cases). On the other hand, when a call raises an exception, it is desirable to have

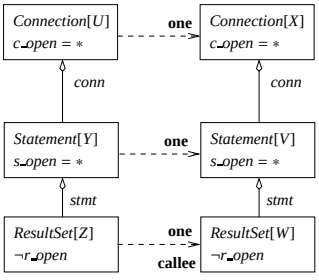


(a) Role mappings.

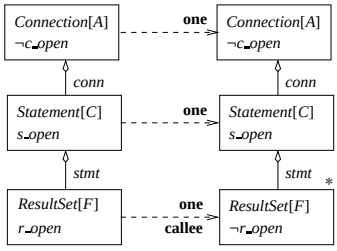


(b) Object mappings. An arrow over a nested subgraph denotes that the nodes of its source are mapped to their isomorphic nodes in the destination.

Fig. 2. The most general rule for `executeQuery`, with no exception.



(a) Closed `ResultSet`.



(b) Closed `Connection`.

Fig. 3. The two most general rules for `next` with `ResultSet not open` exception.

Algorithm 1: *ComputeDPI*.

Input: A set of classes and methods and a set of abstraction predicates
Result: A set of general rules, *Rules*, each of which represents a family of method calls

1 *Rules* = ∅;
2 **while** ¬*Threshold* **do**
3 Pick a snapshot, a concrete Java object, execute one of its methods;
4 Compute, *r*, the corresponding rule of the method call;
5 **if** *there is no* $r' \in Rules$ *that "covers"* r **then** $Rules = Rules \cup \{r\}$;
6 **end**
7 Remove any $r \in Rules$ that is "covered" by another rule;
8 Extrapolate $r \in Rules$ using $r' \in Rules$, when possible; prune rules that are covered by r;
9 Merge all pairs of mergeable rules in *Rules*;
10 Isolate all pairs of similar exception rules in *Rules*;

the smallest rule that isolates the cause of the exception. Furthermore, for exception rules, we use a ternary logic that assigns an unknown value "*" to a predicate of an object when the evaluation of the predicate does not affect whether the exception will be raised or not. These characterizations of the most general rules for a method call are inspired by the monotonic semantics that we have developed for object-oriented programs [6]. For a safe method call, it should be possible to replicate its result in a context with more objects. For a method call with an exception, there is no context with more objects that can avoid the exception.

Figure 3 shows the two rules that our tool computes for the `next` method when it raises the `ResultSet not open` exception. In Figure 3(a), the "*" values for the *s_open* and *c_open* predicates denote that regardless of whether the corresponding statement or connection objects of a `Resultset` object are open or not, the method call over the `Resultset` raises the exception when it is closed. Figure 3(b) shows the case when the `Resultset` is actually open, but its corresponding `Connection` is not. These rules point out succinctly the root cause of a bug discussed in an Apache forum.[1]

2.4 Computation Stages

Creating a rule from a specific method call is only the first step to compute a DPI. Algorithm 1 outlines the main steps that our tool takes to compute succinct DPIs.

The first stage of the algorithm (lines 1-7) is the *exploration stage*, in which a *universal client* non-deterministically explores the behaviour of the package. Each step of the universal client is recorded using a *source* and a *destination snapshot*, each of which is a set of Java objects in the heap. The result of each step of the universal client is a rule. If a new rule is *covered* by another already-explored rule, it is considered redundant and discarded (line 5). Intuitively, a rule r' covers rule r if r' subsumes the behaviour of r by having "larger" elements. The exploration stage continues until a maximum number of redundant rules are encountered. After this threshold is reached, the redundant rules in the set of explored rules are removed (line 7).

[1] `https://issues.apache.org/jira/browse/DERBY-5545`

After the exploration stage, we apply three heuristics to the set of explored rules. Our *extrapolation* heuristic generalizes a rule by expanding its (cast) nested object graphs into more general graphs that represent more heap configurations. Our *merge* heuristic combines a pair of similar rules into one. Similarly, the *exception isolation* heuristic combines a pair of similar exception rules. These heuristics decrease the number of distinct explored rules of a DPI substantially; e.g., in the case of JDBC, from about 2000 distinct rules to 26 final rules.

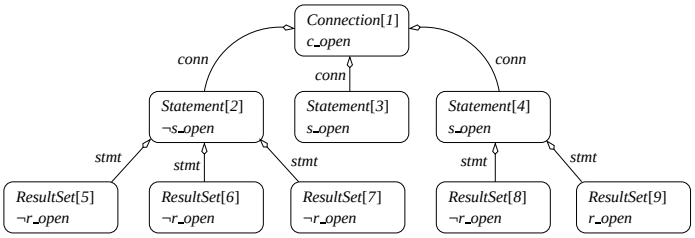
3 Method Calls and Rules

From a Method Call to a Rule. A key step in computing a rule from a method call is to derive the source and destination nested object graphs and cast nested object graphs of a rule from the source and destination snapshots of the method call. The object mapping and role mapping of a rule are simply computed by tracking how objects change from the source to the destination snapshot, and ensuring that if an object is mapped by the role mapping it is not mapped by the object mapping. The computation of nested object graphs is the same for source and destination snapshots, except that a destination snapshot can have newly created objects. For the sake of brevity, at below, we assume that we deal with the source (cast) nested object graph of a rule.

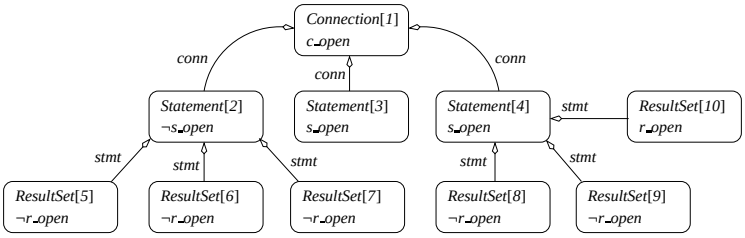
The corresponding snapshot of a cast nested object graph consists of the callee object, actual parameter objects, and all other objects that transitively reach these objects through their references, as well as all objects that are transitively reached from these objects through their references. The corresponding snapshot of a nested object graph consists of all objects in the cast nested object graph plus all objects that can reach these objects transitively. To compute these snapshots, we use the input reference predicates. Next, we describe how to compute a nested object graph.

The first step is to turn the snapshot into a directed labelled graph by using the input scalar and reference predicates. We call such a graph a *heap graph*. Figure 4(a) shows a heap graph corresponding to 9 JDBC objects, using the predicates described in Section 2. Each node of the graph is labelled with the name of its class, the evaluations of its scalar predicates, as well as a unique id that is enclosed inside a pair of brackets. Each edge of the heap graph is labelled with the name of its corresponding reference predicate. Figure 4(b) is another heap graph resulting from the invocation of method `executeQuery` on the Java object that the node with id 4 in Figure 4(a) represents. The nodes with the same identifiers in the two graphs represent the same Java objects.

The second step is to reduce a heap graph to a nested object graph. The idea is that if an object or a pattern for a set of interconnected objects appears more than once, then it is marked as repeatable. The reduction from a heap graph to a nested object graph can be considered as a bisimulation reduction: two nodes in a heap graph are equivalent iff they have the same evaluations for their scalar predicates, and furthermore, they mimic one another by reaching equivalent nodes following their similar reference edges. Figure 5 shows two nested object graphs that our tool computes for the heap graphs in Figure 4. Repetition of a single node is denoted just by a "*" next to it. Repetition of a subgraph (not shown in this figure) is denoted by a dotted line around the subgraph together with a "*"; e.g., as in Figure 2(b). The nodes of the nested object graphs are graphically similar



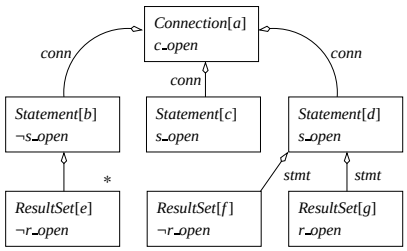
(a) Heap graph before method call.



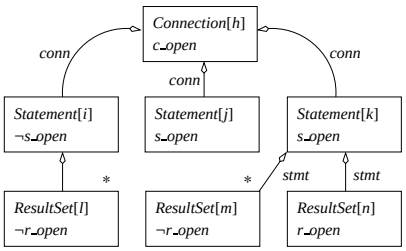
(b) Heap graph after method call.

Fig. 4. Two heap graphs for invocation of `executeQuery` on object 4.

to heap graphs except that they are shown by solid rectangles and they are labelled with alphabetic ids. As examples of repetition, node *e* in Figure 5(a) is the equivalence class for the nodes 5, 6, and 7 in Figure 4(a), and node *m* in Figure 5(b) is the equivalence class for the nodes 8 and 9 in Figure 4(b).



(a) Nested object graph corresponding to heap graph in Figure 4(a).



(b) Nested object graph corresponding to heap graph in Figure 4(b).

Fig. 5. Two nested object graphs.

The computation of a cast nested object graph is similar. The difference is that two objects of the snapshot that have roles cannot be mapped to the same equivalence class.

Rule Coverage Relation. In order to determine whether a rule covers another rule, we need to compare their corresponding (cast) nested object graphs. A nested object graph,

ng, is *subgraph isomorphic* to nested object graph, *ng′* if: (i) *ng* is subgraph isomorphic to *ng′* when their repetition structures are not considered; (ii) the isomorphism relation relates only nodes that have same predicate valuations; and (iii) it is not the case that a node *v′* of *ng′* is not part of a repetition pattern that its corresponding node *v* of *ng* is; i.e., *ng* does not represent a heap configuration that its corresponding subgraph in *ng′* cannot represent. We extend this definition to cast nested object graphs by additionally requiring that only nodes with same role labels can be related by isomorphism.

A rule, *r*, is then *covered* by a rule, *r′*, if: (i) they are both over the same method; (ii) both raise either no exceptions, or the same exception; (iii) the corresponding graphs of *r* are pairwise subgraph isomorphic to the ones of *r′*; and (iv) for each tuple (u, v) of the object mapping of *r* there is a tuple (u', v') in the object mapping of *r′* such that *u* and *u′*, as well as *v* and *v′* are isomorphic; furthermore, it is not the case that the multiplicity of the former tuple is "many" while the multiplicity of the latter tuple is "one"; and (v) similar constraints as iv between the tuples of the role mappings of *r* and *r′*.

4 Generalization Heuristics

4.1 Extrapolation

Sometimes a rule could have covered many other rules if certain nodes in its source and/or destination (cast) nested graphs were marked as repeatable. Our *extrapolation* heuristic could mark such nodes as repeatable using the information in the graphs of other rules.

To identify opportunities for extrapolation, our tool looks for *deficient* nodes in a (cast) nested object graph. A node is deficient if it is not repeated and either the role mapping or the object mapping takes it to a repeated node. Our hypothesis is that a deficient node is not repeated because the exploration did not manage to produce enough objects of that type. For instance, if we consider the graphs in Figure 5 as the source and destination graphs of a rule , f and g, which are both mapped to m, are both deficient nodes. Given a deficient node, our system explores all other rules to find a source or a destination nested object graph into which the corresponding nested object graph of the deficient node can be *embedded* w.r.t. the subgraph isomorphism relation. If according to the embedding the node corresponding to the deficient node in the other graph is repeated, then the deficient node will be marked as repeatable too. In our example, our tool can find an embedding relation that leads to the extrapolation of f. However, g cannot be extrapolated. Indeed, each JDBC `Statement` object cannot have more than one open `ResultSet` object.

Repetition is propagated to all nodes pointing to the extrapolated node, in order to ensure that there is no non-repeated node pointing to a node that is marked as repeatable. Lastly, the multiplicities of mappings might need to be adjusted to ensure that a node that is marked as repeatable is not mapped only once via a "one" multiplicity. The extrapolation heuristic is applied to all rules after the exploration stage, and then all redundant rules are removed.

4.2 Merging

While the extrapolation stage prunes a substantial number of rules, there may still be a large number of rules in a DPI, e.g., thousands of rules for JDBC. The reason is that different rules for the same method might have explored different instances of heaps that have incomparable sets of objects, and there are various exception cases. To further reduce the number of the rules, we have developed the *merging* heuristic, which combine sets of related rules into one.

To check whether two rules can be merged, we compare a part of their cast nested object graphs that we call the *upward* part. The upward part of a cast nested object graph is its subgraph that consists of the set of nodes that are labelled by roles plus the nodes that are reached from these nodes. A pair of rules are *mergeable* if: (i) the upward parts of their source and destination cast nested object graphs are pairwise isomorphic; and (ii) their role mappings restricted to the upward parts are similar and over isomorphic nodes. For a mergeable pair of rules, the merge heuristic essentially first computes their union and then performs a reduction over the resulting source and destination nested object graphs of the resulting rule. The reduction replaces a nested object graph with its smallest subgraph that simulates all other subgraphs of the original graph. This reduction is in the spirit of *downward closed* graphs where a nested object graph not only represents all heap instances arising from the repetition of its repeatable subgraphs, but also represents any graph which is a subgraph of those – hence the term "downward closed" [5]. Finally, the role mapping and object mapping of the resulting rule are adjusted according to the reduction. As an example, assuming that the nested object graphs in Figure 5 belong to a rule, then node *c* in Figure 5(a), for instance, would be mapped to node *C* in Figure 2(b) during the merge operation. Similar to the extrapolation heuristic, the multiplicities of mappings might need to be adjusted.

4.3 Exception Isolation

While the merge heuristic corresponds to the union of a set of rules, the *exception isolation* heuristic corresponds to the intersection of a set of exception rules. This heuristic deals only with the cast nested object graphs; the nested object graphs are discarded. For a pair of rules that raise the same exception and whose cast nested object graphs are isomorphic when their scalar abstraction predicates are not considered, this heuristic essentially combines the corresponding nodes of the cast nested object graphs of the two rules via a ternary logic. If the values of a predicate are different, the unknown value, denote by "*", is chosen. Nested object graphs of the rules are not useful because often when an exception is raised the states of the corresponding objects of these graph do not change. Furthermore, we are interested in identifying the smallest contexts in which an exception can raise.

5 System

Figure 6 shows the high-level architecture of our system, implemented in Java.The arrows specify the high-level information communicated between the components.

Fig. 6. The main components of the system.

The *Package Abstraction* component provides the information about the input package. It consists of a set of classes whose methods provide the names of classes of the package under study, their methods, and predicate abstractions. These classes use Java reflection to obtain these information. Furthermore, there are classes that provide the actual parameters for the method calls of the universal client; these parameters have random values.

The *Package Explorer* component implements the exploration stage of Algorithm 1. To implement the snapshots whose objects can be accessed throughout the exploration, our tool maintains the corresponding trace of method calls that resulted in the snapshot. To call a method of an object of a snapshot, our tool recreates the entire snapshot by replaying its corresponding trace. Cloning or saving an object, in general, would not work, as not all classes implement these methods. A recreated snapshot has similar objects as the original snapshot, assuming that, as far as the abstraction predicates are concerned, method calls are deterministic. To relate the objects in a snapshot to the objects in its replayed copy, we use a notion of *logical id* for each of the objects of the snapshots; objects that have the same logical ids are treated as copy of one another.

To ensure that our exploration does not prematurely identify objects as non-repeatable in a rule, we use a repetitive object creation scheme in our exploration: if a creator method is chosen to be executed, we invoke the method $n > 1$ number of times consecutively, and only after that compute the rule with respect to the snapshot before consecutive method calls and the snapshot after that. Also, after the initial exploration stage, to achieve a good coverage, similar to other approaches [3], our system ensures that all possible method calls on all objects of all rules in the repository are executed and their corresponding rules are stored in the repository.

The *Heuristics* component implements the algorithms in Section 4. We use the graph data structures in the JGraphT library to implement our graph algorithms.

Limitations. While we expect our tool to work in a straightforward manner on packages that solely work on the heap (e.g., Java collections), for packages that work with external components, the Package Abstraction part is more complex, because an environment needs to be set up. Also, the feasibility of the replay mechanism should be considered. These limitations are inherent to dynamic approaches.

6 Experiences

We have used our tool to compute the DPI of three Java packages: `JDBC`, `ArrayList`, and `HashSet`. While our tool usually identifies the expected set of rules for the DPI, some of these rules could, in principle, be more general. The converse, however, has never happened in our experiments. A rule computed by our tool always corresponded to an actual behaviour of the package.

Table 1. Duration and number of rules after different stages in computing DPIs of three packages. Information, except for the last column, correspond to average values of five runs.

Package	Threshold #	Time (min:sec)				#Rules			
		Exploration	Extrapolation	Merging	Isolation	Exploration	Extrapolation	Merging	Isolation
`ArrayList`	200000	010:37	000:03	000:00	000:00	572	299	29	15 (once 14)
`HashSet`	200000	168:26	000:23	000:01	000:00	1140	503	34	16
`JDBC`	1200	032:01	000:57	000:05	000:00	2465	2370	29	26 (twice 25)

Table 1 shows the results of our experiments for each of these packages. The measurements for each package are for the average of five runs on a dual-core CPU Windows 7 desktop machine with 8 GB of RAM. In all our experiments, we have set JVM options to use 5GB of physical memory. For each package, Table 1 presents the time taken and the number of rules after each stage of the computation, namely after the exploration, extrapolation, merge, and exception isolation phases.

JDBC. In Section 2, we already presented some of the rules of the DPI of JDBC. In our experiments, the universal client connects to a local Apache Derby database. We use a key-value table that is manipulated through INSERT, DELETE, and SELECT SQL commands with random values, via JDBC. We are thus assuming that the DPI of the JDBC package is independent of the schema of databases to which it connects. This is justified by our interest in determining the relationship of interacting objects of a package, and not its interaction with external components. Increasing the threshold value to larger than 1200 would cause out-of-memory exceptions. Our tool computed 26 rules in three out of five runs; in the other two runs, it computes 25 rules. The missing rule in both cases was the rule for the `close` method when called over an open `ResultSet` that is connected to a closed `Statement` and a closed `Connection`.

ArrayList. We consider two classes of `ArrayList`: `Array` and its internal class `Itr`, which implements Java `Iterator`. Besides the methods of these classes that create objects, we consider the `Add` method of `Array`, and the `next` and `remove` methods of `Itr`. We provide a reference predicate, $iter\_of$, to the system, denoting which `Itr` object belongs to which `Array` object. We provide four scalar predicates to the system: $empty \equiv size > 0$, which determines whether an `Array` object is empty or not, $nextCalled \equiv lastRet \neq -1$, which determines whether the `remove` method of an `Itr` object can be called (i.e., if `next` has been called), $mover \equiv size > cursor$, which determines whether an `Itr` has traversed all members of its corresponding `Array` or not, and $sync \equiv modCount = expectedModCount$, which determines whether an `Array` object and an `Itr` object agree on their version numbers (i.e., if the `Array` object has been modified by another `Itr` object). Lastly, we use integers as the domain of `Array`.

Our tool computed 15 rules that cover all possible behaviour of `ArrayList`. It once missed computing the rule for `next` when called on an iterator whose all predicates are true and remain true after the method call. Figure 7 shows the object mapping of one of the three rules that our tool computes for the `remove` method in one of our experiments. The role mapping, not shown here, changes only the *nextCalled* predicate of the callee

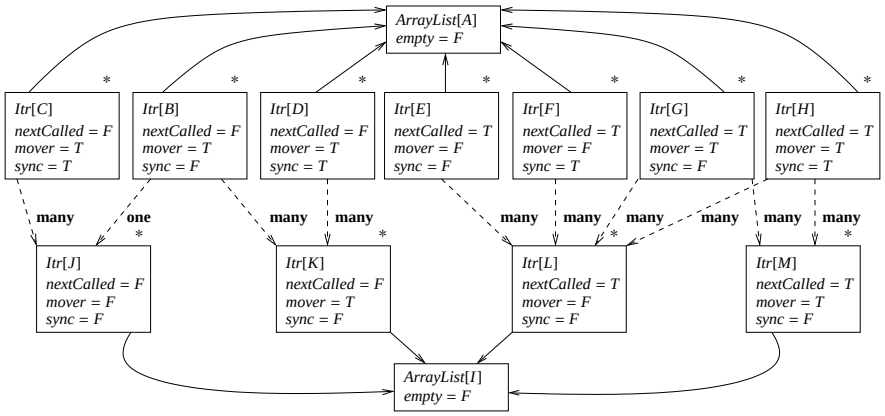


Fig. 7. The object mapping of a rule for `remove` method of `ArrayList`. "*T*" and "*F*" represent *true* and $false$, respectively. For clarity, the reference edges are not labelled with *iter_of*.

iterator object whose all scalar predicates are true. This rule is interesting because it demonstrates that the object mapping of a rule can be non-deterministic. The rule could have been more general, however. First, in the source nested object graph, the object with *nextCalled = false*, *mover = false*, and *sync = false* is missing. Second, the object mapping from *B* to *J* could have had multiplicity "many". And lastly, there could have been an object mapping from *D* to *L* with multiplicity "many" denoting that some of the mover, sync objects whose *nextCalled* is false become non-movers.

HashSet. The DPIs of `HashSet` and `ArrayList` are computed using similar predicates, but `HashSet` uses a `HashMap` class internally, instead of a resizable array. The DPIs of two package are also somewhat different. The main difference is that the `add` method of `HashSet` does not change the heap if its input parameter is duplicate; thus, there is an extra rule that captures this behaviour. Another difference is that the *mover* predicate of an `Iterator` object of a `HashSet` only correctly denotes whether it has traversed all elements of its corresponding `HashSet` if its *sync* predicate is true. This is because unlike an `ArrayList` object, whose iterator objects maintain an index of the underlying array of the `ArrayList` object, the iterators of a `HashSet` objects needs to traverse the underlying hash table of its internal `HashMap` object. Lastly, computing the DPI of `HashSet` takes significantly longer than `ArrayList`'s, both because of their different underlying data structures and because significantly more reflections are needed when evaluating the abstraction predicates of `HashSet`.

7 Conclusion

We have introduced the notion of dynamic package interfaces (DPI). DPIs provide a succinct way to describe valid usage patterns for a package. The DPI of a package is a set of rules, each of which specifies the effect of a method call over a general configuration of a set of objects. We have developed a dynamic tool that computes an approximation of the DPI of a Java package automatically, given a set of abstraction predicates.

The rules of such a DPI generalize the usual examples used in the documentation of the Java package and can be traced to problems discussed in online forums.

A DPI captures both the *inter*-object aspects of the dynamic behaviour of the classes of a package, as well as the *intra*-object aspects of individual classes of the package, relative to a set of scalar and reference predicates, even when unboundedly many objects interact.[2] In contrast, previous dynamic techniques primarily focus on either deriving intra-object specifications for one object or deriving finite state machines that capture the interaction pattern of a finite number of objects [3, 7, 8, 11–13].

References

1. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.K.: General decidability theorems for infinite-state systems. In: LICS '96. pp. 313–321. IEEE (1996)
2. Alur, R., Černý, P., Madhusudan, P., Nam, W.: Synthesis of interface specifications for Java classes. In: POPL'05. pp. 98–109. ACM (2005)
3. Dallmeier, V., Knopp, N., Mallon, C., Hack, S., Zeller, A.: Generating test cases for specification mining. In: ISSTA. pp. 85–96. ACM (2010)
4. Damm, W., Harel, D.: LSCs: Breathing life into message sequence charts. Formal Methods in System Design 19(1), 45–80 (2001)
5. Esmaeilsabzali, S., Majumdar, R., Wies, T., Zufferey, D.: Dynamic package interfaces - extended version. CoRR abs/1311.4934 (2013)
6. Esmaeilsabzali, S., Majumdar, R., Wies, T., Zufferey, D.: A notion of dynamic interface for depth-bounded object-oriented packages. CoRR abs/1311.4615 (2013)
7. Ghezzi, C., Mocci, A., Monga, M.: Synthesizing intensional behavior models by graph transformation. In: ICSE. pp. 430–440. IEEE (2009)
8. Henkel, J., Reichenbach, C., Diwan, A.: Discovering documentation for java container classes. IEEE Trans. Software Eng 33(8), 526–543 (2007)
9. Henzinger, T., Jhala, R., Majumdar, R.: Permissive interfaces. In: Wermelinger, M., Gall, H. (eds.) ESEC/SIGSOFT FSE. pp. 31–40. ACM (2005)
10. Nanda, M., Grothoff, C., Chandra, S.: Deriving object typestates in the presence of inter-object references. In: OOPSLA. pp. 77–96. ACM (2005)
11. Nguyen, T.T., Nguyen, H.A., Pham, N.H., Al-Kofahi, J.M., Nguyen, T.N.: Graph-based mining of multiple object usage patterns. In: ESEC/SIGSOFT FSE. pp. 383–392. ACM (2009)
12. Pradel, M., Jaspan, C., Aldrich, J., Gross, T.: Statically checking API protocol conformance with mined multi-object specifications. In: ICSE'12. pp. 925–935. IEEE (2012)
13. Pradel, M., Gross, T.R.: Automatic generation of object usage specifications from large method traces. In: ASE. pp. 371–382. IEEE Computer Society (2009)
14. Strom, R.E., Yemini, S.A.: Typestate: A programming language concept for enhancing software reliability. IEEE Transactions on Software Engineering 12(1), 157–171 (Jan 1986)
15. Wasylkowski, A., Zeller, A.: Mining temporal specifications from object usage. Autom. Softw. Eng. 18(3-4), 263–292 (2011)
16. Whaley, J., Martin, M., Lam, M.: Automatic extraction of object-oriented component interfaces. In: ISSTA. pp. 218–228 (2002)
17. Wies, T., Zufferey, D., Henzinger, T.: Forward analysis of depth-bounded processes. In: FOSSACS. LNCS, vol. 6014, pp. 94–108. Springer (2010)

[2] We use the terms "inter-object" and "intra-object" in a similar sense as in OO design [4].