Verifying Safety Properties using Separation and Heterogeneous Abstractions

Eran Yahav[*]
School of Computer Science
Tel-Aviv University
Tel-Aviv, 69978 Israel
yahave@post.tau.ac.il

G. Ramalingam
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598 USA
rama@watson.ibm.com

ABSTRACT

In this paper, we show how *separation* (decomposing a verification problem into a collection of verification subproblems) can be used to improve the efficiency and precision of verification of safety properties. We present a simple language for specifying *separation strategies* for decomposing a single verification problem into a set of subproblems. (The strategy specification is distinct from the safety property specification and is specified separately.) We present a general framework of *heterogeneous abstraction* that allows different parts of the heap to be abstracted using different degrees of precision at different points during the analysis. We show how the goals of separation (i.e., more efficient verification) can be realized by first using a separation strategy to transform (instrument) a verification problem instance (consisting of a safety property specification and an input program), and by then utilizing heterogeneous abstraction during the verification of the transformed verification problem.

Some tasks are best done by machine,
while others are best done by human insight;
and a properly designed system will find the right balance.
– D. Knuth

General Terms

Algorithms, Languages, Theory, Verification

Keywords

Abstract Interpretation, Verification, Safety Properties, Program Analysis, Typestate Verification

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

[*]This research was supported in part by a grant from the Ministry of Science, Israel, and the Israeli Academy of Science, and was done in part while the author was visiting IBM Research.

1. INTRODUCTION

Recently there has been significant and growing interest in static verification of safety properties (e.g., see [3, 6, 2, 9, 8, 1, 14, 7, 5]). Such verification is valuable since it can identify software defects early on, thereby improving programmer productivity, reducing software development costs, and increasing software quality and reliability.

Consider the Java program fragment shown in Fig. 1. This program performs a number of database queries using JDBC [21]. This example violates one of the usage constraints imposed by the JDBC library. Specifically, the execution of a query in line 28, using a `Statement` object, has the implicit effect of discarding the results to the previous query executed in line 27 (using the same `Statement` object). Hence, the subsequent attempt to use these discarded results, in line 40, is invalid.

We are interested in verifying that a given program satisfies safety properties of the kind illustrated above. While significant progress has been made recently in such lightweight verification, doing precise verification that can scale to large and complex programs still remains a challenge. In this paper, we investigate a technique to improve the precision and efficiency of such verification.

The starting point for our work is the notion of *separation*: the idea that separating or decomposing a verification problem into a collection of smaller subproblems can help scale verification algorithms (e.g., see [5]). Consider again the example in Fig. 1. This example program executes 5 different queries, producing 5 different `ResultSet`s. We can verify that the program satisfies the desired safety property by *independently* verifying the property for each of these `ResultSet`s.

It may seem like we are just restating the problem, but this restatement is important from the point of view of the underlying analysis. It can significantly increase the efficiency of the analysis by reducing the size of the state space that needs to be explored. In our running example, `Statement stmt1` and `ResultSet rs1` can be in several possible states in line 28. While this information is relevant for verifying subsequent use of `ResultSet rs1`, it is irrelevant for verifying the usage of `ResultSet rs2`, for example. The motivation for separation is to exploit this to improve efficiency, without losing precision.

In this paper, we explore this approach by addressing the following questions:

(1) How do we decompose a verification problem into a collection of subproblems?

```
…
10  ConnectionManager cm = new ConnectionManager();
11  Connection con1 = cm.getConnection();
12  Statement stmt1 = cm.createStatement(con1);
…
15  ResultSet maxRs = stmt1.executeQuery(maxQry);
16  if (maxRs.next())
…
18  ResultSet rs1 = stmt1.executeQuery(balancesQry);
19  if (maxBalance1 < threshold) {
20    stmt1.close();
21    closed1 = true;
22  }
```

```
23  Connection con2 = cm.getConnection();
24  Statement stmt2 = cm.createStatement(con2);
…
27  ResultSet rs2 = stmt2.executeQuery(balancesQry);
28  ResultSet maxRs2 = stmt2.executeQuery(maxQry);
29  if (maxRs2.next())
…
31  ResultSet minRs2 = stmt2.executeQuery(minQry);
…
40      while (rs2.next())
…
```

Figure 1: JDBC example snippet.

(2) How can we adapt the state abstractions to each subproblem (so that we may achieve the desired efficiency improvement)? One of the key characteristics of our approach is that we break up this question into two parts: (a) What are the objects that are *relevant* to a verification subproblem? (b) Given the set of relevant objects, how can we *adapt* the state abstraction to utilize this information?

In this paper, we introduce the notion of a *separation strategy* as something that can help answer question (1) and partly help answer (2)(a). Rather than adopt a fixed strategy for separation, we introduce a simple language for specifying separation strategies that can be used to manually specify strategies. One strategy for the JDBC problem would be to apply separation at the level of a `Connection`, where verification of all `ResultSets` created over a single `Connection` is treated as a single verification subproblem.

Currently, we see the strategy specification language as a way for analysis designers, such as ourselves, to specify and experiment with different strategies. Our intuition, however, is that end users may be able to easily identify objects of interest and relevance to some verification subproblem and that the strategy specification may be a lightweight way to allow end user input to guide verification.

Given a verification problem instance (consisting of a safety property specification and an input program) and a separation strategy, the first step of our approach is to *transform* (or instrument) the verification problem instance to reflect the separation strategy. (Here, it is worth pointing out that when we talk about "decomposing a verification problem into subproblems", we are talking at a conceptual level; the transformed verification problem mentioned above is equivalent to solving the subproblems in parallel.)

The second step is to perform verification for the transformed program and safety property in a way that exploits the separation. This leads us to question (2) above. One of the distinguishing characteristic of our approach is that we rely on an *integrated* analysis that performs, e.g., heap analysis in conjunction with the verification (as opposed to doing it as a separate preceding analysis). Thus, we are interested in exploiting *separation* even for the heap analysis. (Indeed, the benefits of separation may be greatest for the heap analysis component if the verification utilizes precise, but expensive, heap analysis.)

In this paper, we utilize *heterogenous abstractions* that allow us to model different parts of the heap with different degrees of precision at different points in time as a technique to exploit separation.

Consider the example in Fig. 1. Fig. 2(a) informally shows two possible states of the heap at line `28`, corresponding to different branches taken at line `19`. The `Statement` referenced by `stmt1` and the `ResultSet` references by `rs1` are in a *closed* state in C_2 (as illustrated by the "c" inside the component node). Fig. 2(b) illustrates the abstract representation produced by our technique (with a simple separation): the representation above the line corresponds to one subproblem (corresponding to `Connection con1`), and

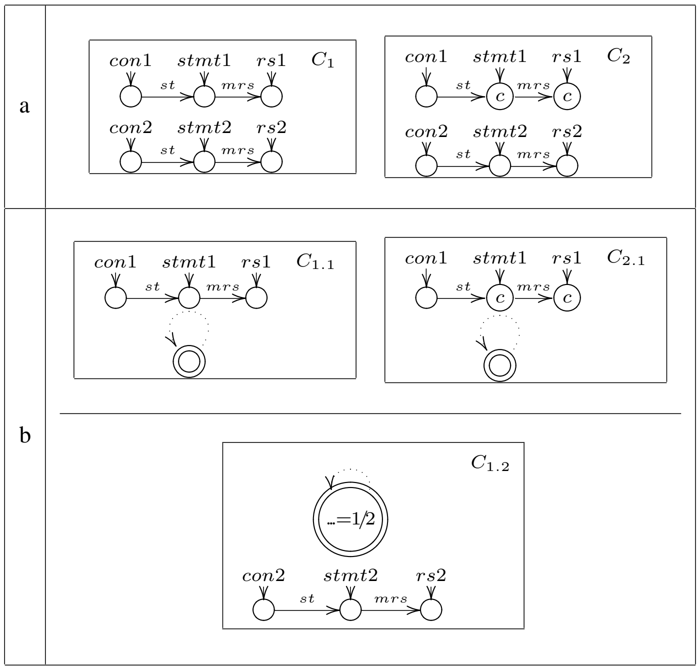

Figure 2: Separation and heterogenous abstraction.

the representation below the line corresponds to a different subproblem (corresponding to `Connection con2`). (We present more details about these representations in later sections.)

Main Results

The main contributions of this paper are:

- We present a simple language for specifying separation strategies for decomposing a single verification problem into a set of subproblems.
- We present a general framework of *heterogeneous abstractions* that allows different parts of the heap to be abstracted using different degrees of precision at different points during the analysis.
- We show how the goals of separation (i.e., more efficient verification) can be realized by first using a separation strategy to transform (instrument) a verification problem instance (consisting of a safety property specification and an input program), and then utilizing heterogeneous abstraction during the verification of the transformed verification problem.
- We have implemented a prototype of a separation verification engine using TVLA, and applied it to verify properties of several Java programs, using several different separation

```
while (?)  {
   f = new File();
   f.read();
   f.close();
}
```

Figure 3: Program illustrating the difficulty of verifying that a file component is never read after it has been closed.

strategies. Initial results indicate that separation does improve the efficiency, and possibly precision, of verification results.

One of the themes to emerge in recent work (e.g., see [14, 7, 5]) is that maintaining just the right correlation required between "analysis facts" can be the key to efficient and precise verification: maintaining no correlations (independent attribute analysis) can lead to imprecision, while maintaining all correlations (relational analysis) can lead to inefficiency. However, finding this intermediate ground can be hard for heap analyses that, e.g., use graph-based representations of the heap. Our approach may be seen as a step towards achieving such a balance in a heap representation.

Existing approaches to verification range from more automated techniques that rely on no extra human input (other than the safety property specification) to techniques that rely on end users to provide significant annotation, such as program invariants. We see the strategy specifications we use as a potentially useful, lightweight, way for users to assist a verifier.

Related Work

ESP [5] is a system for typestate verification [19] that utilizes a simple fixed separation technique. Our work differs from ESP in several respects. ESP uses a two-phase approach to verification in which pointer-analysis is performed first, followed by typestate verification. Often, this prevents ESP from applying "strong" updates necessary for successful verification. Separation in ESP is exploited only in the typestate verification phase. We utilize an integrated analysis, where the heap analysis and verification are performed simultaneously, allowing the heap analysis to benefit from separation. We also explore separation in a more general setting than ESP: we explore its applicability to first order safety properties, such as the ones shown earlier for JDBC, which involve relationships among multiple objects; we allow user-specifiable separation strategies; finally, our technique can achieve separation between multiple objects allocated at the same allocation site. Since our analysis is capable of separating out a *single* object (even from among multiple objects allocated at the same allocation site), it can utilize "strong" updates when ESP is forced to use "weak" updates. This can lead to more precise results, as illustrated by the example in Fig. 3. Unlike ESP, our technique can successfully verify this example.

The instrumentation technique we use to implement separation strategies may be seen as an extension of techniques previously used (e.g., by Bandera [3, 4] and SLAM [13]) to instrument a program with respect to a safety property specification prior to verification. However, these approaches use such instrumentation purely to encode the verification problem, and do not exploit it for separation and the generation of adaptive abstractions like we do.

Separation is similar in spirit to McMillan's functional decomposition [12], which divides the verification task according to units-of-work rather than dividing according to the program syntax. His division, however, is applied at the specification level since all entities have static names.

Guyer [10] shows that it is valuable to have pointer analyses that are client-driven. His analysis is a two pass analysis, with a client-independent first pass pointer analysis, followed by a second pass pointer analysis that uses different levels of context-sensitivity for different analyzed procedures, based on sources of imprecision identified from the use of the results computed by the first pass.

[14] explores techniques to derive abstractions that are specialized to a safety property. Our work on separation is orthogonal to these techniques. In [18], a heap-safety-automaton (HSA) is used to specify local heap properties (corresponding to typestate properties) which are later verified without using any form of separation. We believe that the separation techniques in this paper could be beneficial for their analysis as well.

Our heterogeneous abstraction technique is based on the parametric analysis framework of Sagiv et al. [17]. This analysis framework has been used to derive several powerful and precise, but very expensive, heap analyses. We believe that successful verification systems need to use such powerful analyses when needed (to handle difficult cases when they arise), but scalability requires that the scope of such analyses be restricted to a small enough universe. We believe that the identification of "relevant" objects via our separation technique is a step towards achieving this.

An alternative separation technique would be to decompose a verification problem into subproblems that verify that each *use* of an object, such as a `ResultSet`, is safe, utilizing demand-driven analysis to solve the subproblems. This inherently involves "backward analysis", while our approach utilizes "forward analysis". The motivation for our approach is that "backward analysis" is inherently hard when complex heap analysis is involved.

2. SAFETY PROPERTIES

We are interested in verifying that client programs that use a component (library) satisfy correct usage constraints imposed by the library API. In this paper, we use some of the usage constraints imposed by the JDBC library to illustrate our separation technique for verification of such safety properties.

The JDBC library allows client programs to create `Connections` to databases. Any number of `Statements` may be created over a `Connection`. A `Statement` can be used to execute an SQL query over the database, via the `executeQuery()` method, which returns the results to the query as a `ResultSet`. The `next()` method of a `ResultSet` can be used repeatedly to iterate over the results of the query. However, the execution of the `executeQuery()` method of a `Statement` implicitly *closes* any `ResultSet` previously returned by the `Statement`, and it is invalid to use any of those `ResultSets` any more. Similarly, after closing a `Connection`, it is invalid to use any of the `Statements` created from that `Connection` or any of the `ResultSets` returned by these `Statements`.

Thus, the execution of line 28 in the example of Fig. 1 implicitly closes the `ResultSet` created in line 27, and this will cause an error when this closed `ResultSet` is used in line 40.

We specify safety properties using `Easl` [14], a procedural language for specifying an abstract semantics for a component library. `Easl` statements are a subset of Java statements containing assignments, conditionals, looping constructs, and object allocation. `Easl` types are restricted to booleans, heap-references, and a built-in abstract Set and Map types. Finally, `Easl` provides a `requires` statement to specify the correct usage constraints imposed by the library: it is the responsibility of any program that uses the library to ensure that the condition specified by the `requires` clause will hold true at the corresponding program point. These are the safety properties we are interested in checking.

`Easl` supports object references and dynamic allocation. This allows us to naturally express the structural relationships between the objects of interest, as well as dynamic allocation of these objects.

```
class Connection {
  boolean closed;
  Easl.Set statements;
  Connection() {
    closed = false;
    statements = {};
  }
  Statement createStatement() {
    requires !closed;
    Statement st = new Statement(this);
    statements = statements U { st };
    return st;
  }
  void close() {
    closed = true;
    for each st in statements
      if (st.myResultSet != null) {
        st.closed = true;
        st.myResultSet.closed = true;
      }
  }
}
```

```
class Statement {
  boolean closed;
  ResultSet myResultSet;
  Connection myConnection;
  Statement(Connection c) {
    closed = false;
    myConnection = c;
    myResultSet = null;
  }
  ResultSet executeQuery(String qry) {
    requires !closed;
    if (myResultSet != null)
      myResultSet.closed = true;
    myResultSet = new ResultSet(this);
    return myResultSet;
  }
  void close() {
    closed = true;
    if (myResultSet != null)
      myResultSet.closed = true;
  }
}
```

```
class ResultSet {
  boolean closed;
  Statement ownerStmt;
  ResultSet(Statement s) {
    closed = false ;
    ownerStmt = s;
  }
  void close() {
    closed = true;
  }
  boolean next() {
    requires !closed;
  }
}
```

Figure 4: An `Easl` specification for a simplified subset of the JDBC API.

Fig. 4 shows an `Easl` specification for the JDBC[1] safety properties described above.

Note the use of the set `statements` and the fields `myResultSet`, `myConnection`, and `ownerStmt` to specify the relationships between the components. Also note that applying `executeQuery` closes the `ResultSet` component referenced by `myResultSet` if one exists.

In the rest of this paper we will address the problem of verifying that a given Java program satisfies the safety properties specified by an `Easl` specification.

3. SEPARATION STRATEGIES

The goal of a separation strategy is to separate or decompose a verification problem into a collection of verification subproblems. We now present an informal description of separation strategies. A more formal meaning will be given to separation strategies in Sec. 4.2.

Consider a typestate property, such as "an `InputStream` should not be `read` after it is `closed`". In this case, verification of the safety property for one `InputStream` object does not depend on the state of another `InputStream` object. Hence, the verification can be done independently for each `InputStream` object. This amounts to a very simple separation strategy.

Some safety properties, such as the `JDBC ResultSet` property, involve multiple related objects – we refer to these as *first order safety properties*. Consequently, verification of such properties can be separated into subproblems in several different ways, each with potentially different efficiency and precision tradeoffs. Before we present some of the possible separation strategies, we introduce a simple language for specifying a separation strategy.

In our approach, a separation strategy represents a method for *choosing* a set of objects. A set of chosen objects identifies a subproblem where verification is restricted to the chosen objects. For effective verification, a strategy should identify other objects that may have an impact on a chosen object and choose them too. This motivates the definition of the following language for specifying strategies.

An (atomic) separation strategy is a sequence of *choice* operations, where each choice operation identifies one or more objects that are chosen, as a function of previously chosen objects.

```
<atomic-strategy> ::= <choice-spec> *
```

[1]Field names from Sun's SDK1.3.1 sun.jdbc.odbc implementation.

```
<choice-spec> ::=
  choose (some|all) <var>:<constr> [/<condition>]
<constr> = <type-name> ( <var-list> )
```

Each choice operation consists of a variable name, a signature of a constructor, and an optional condition. The choice operation `choose some` performs a non-deterministic selection of objects created through the specified constructor that satisfy the condition. The operation `choose all` chooses all objects created through the specified constructor that satisfy the condition. Both choice operations evaluate the condition, and apply their choice on entry to the specified constructor.

We now present some strategies for the `JDBC ResultSet` property.

Single Choice. The motivation for our first strategy is the observation that there is no interaction between different `Connections`: it should be possible to perform verification for each `Connection` independently. Hence, the following strategy performs separation at the level of a `Connection`.

```
choose some c :  Connection()
choose all s :  Statement(x) / x == c
choose all r :  ResultSet(y) / y == s
```

The separation strategy described above first non-deterministically chooses a single `Connection`, then proceeds by choosing *all* `Statements` created from this `Connection`, and then choosing *all* `ResultSets` created from these `Statements`. For the running example, this amounts to separating the verification problem into two independent subproblems, one for each `Connection`.

Multiple Choice. However, it should be clear from the JDBC specification that it is possible to perform a more fine-grained separation than the single choice strategy described above. In particular, the correct usage of a `ResultSet` does not really depend on how *any* other `ResultSet` is used. Thus, it is not necessary to perform verification of the different `ResultSets` created from a single `Statement` together, for instance. However, the correct usage of a `ResultSet` does depend on the `Statement` and `Connection` underlying the `ResultSet`. These observations motivate the following separation strategy.

```
choose some c :  Connection()
choose some s :  Statement(x) / x == c
choose some r :  ResultSet(y) / y == s
```

For the running example, this strategy produces a set of 5 subproblems, one for each combination of matching Connection, Statement and ResultSet.

Note that using a finer grained separation strategy may or may not lead to more efficient verification. On one hand, finer grained separation leads to smaller subproblems that can be verified more easily. On the other hand, it also leads to a larger number of subproblems. The relative performance of a strategy may depend on the amount of work that is duplicated across the different subproblems. The strategy we present next is likely to reduce the amount of work duplicated across subproblems.

Incremental. The two strategies we have seen are examples of *atomic* strategies. In this paper, we also explore the possibility of applying a sequence of increasingly complex separation strategies to perform verification. The motivation for this is simple: usually many verification subproblems may be amenable to simple and efficient verification, but some verification subproblems may require more precise analysis for successful verification.

An incremental strategy is a sequence of atomic strategies, which are tried one after another, stopping when one of the atomic strategies completely verifies the program. An atomic strategy can make use of failure information from the previous atomic strategy applied to the program. We restrict ourselves to a very simple form of failure information, where the choice operation can restrict attention to individuals that failed verification in the previous step. We will illustrate this with examples first, and later explain how these strategy specifications are interpreted.

```
{
  choose some r :  ResultSet(y)
} on failure {
  choose some s :  Statement(x)
  choose some failing r :  ResultSet(y) / y == s
} on failure {
  choose some c :  Connection()
  choose some failing s :  Statement(x) / x == c
  choose some failing r :  ResultSet(y) / y == s
}
```

The above strategy optimistically first attempts to verify usage of each ResultSet independent of even the Statement underlying the ResultSet. If that fails, it then attempts to verify usage of ResultSets, while tracking usage of the underlying Statement. If that too fails, it then attempts verification using even more context.

Note that an incremental strategy may be thought of as a very simple (fixed) iterative refinement scheme. For our running example, the very first atomic strategy in the sequence above successfully verifies all correct uses of ResultSet.

Semantics and Correctness. Note that the language presented above is powerful enough to specify *partial* verification problems, where the checking is done only for the specified *subset* of objects. This power is useful in some contexts. However, the goal of a *strategy* is typically to improve the precision and efficiency of verification but not affect its correctness. In order for a separation strategy to guarantee correctness, it has to *cover* all objects of the types being verified.

We later describe how a strategy specification defines an instrumented semantics for a program: every program-state in the standard semantics corresponds to a set of instrumented-program-states in the instrumented semantics, where an instrumented-program-state may be roughly thought of as a program-state plus a set of objects in the program-state (which are the "chosen" objects). A strategy is said to completely cover a type T if for every program-state σ in the standard semantics, for every object *obj* of type T in σ, there exists an instrumented-program-state in which *obj* is a chosen object.

Predicates	Intended Meaning
$x(v)$	reference variable x points to the object v
$fld(v_1, v_2)$	field f of the object v_1 points to the object v_2
$bv()$	boolean variable bv has true value
$bf(v)$	boolean field bf holds for object v
$site[AS](v)$	object v was allocated in allocation site AS

Table 1: Predicates for partial Java semantics.

Theorem 1. *A separation strategy that consists only of choice operations with no condition and choice operations of the form:*

```
choose all x :  T (w_1,...,w_k) / (w_i == z_j)
```

where $\mathtt{w}_i$ *(*$1 \leq i \leq k$*) is a parameter of the constructor* T*, and* $\mathbf{z}_j$ *is a variable bound by earlier choice operations, completely covers* T*.*

4. SEPARATION

In this section, we show how a separation strategy is utilized to decompose a verification problem into a set of verification subproblems. We first illustrate how an Easl safety property specification and a Java program together can be translated into an analysis problem instance in the parametric analysis framework of [17]. We then show how an Easl safety property specification, a Java program, and a separation strategy specification together can be translated into a *modified* analysis problem instance (corresponding to a set of verification subproblems). (This translation provides the semantics of a separation strategy.)

4.1 Background

We now present an overview of *first order transition systems* (FOTS), the formalism underlying the parametric analysis framework of [17]. FOTS may be thought of as an imperative language built around an expression sub-language based on first-order logic

In a FOTS, the state of a program is represented using a first-order logical structure in which each individual corresponds to a heap-allocated object and predicates of the structure correspond to properties of heap-allocated objects.

Definition 1. A 2-valued logical structure over a set of predicates P is a pair $C^\natural = \langle U^\natural, \iota^\natural \rangle$ where:

- $U^\natural$ is the universe of the 2-valued structure. Each individual in $U^\natural$ represents a heap-allocated object.
- $\iota^\natural$ is the interpretation function mapping predicates to their truth-value in the structure: for every predicate $p \in P$ of arity k, $\iota^\natural(p) : {U^\natural}^k \to \{ 0, 1 \}$.

In the following we will use $p(v)$ as shorthand for $\iota^\natural(p)(v)$ when no confusion is likely.

Table 1 shows some of the predicates we use to record properties of individuals in this paper. A unary predicate $x(v)$ holds when the reference (or pointer) variable x points to the object v. Similarly, a binary predicate $fld(v_1, v_2)$ records the value of a reference (or pointer-valued) fieldfld. A nullary predicate $bv()$ records the value of a local boolean variable bv and a unary predicate $bf(v)$

Predicates	Intended Meaning
$chosen[x](v)$	object v was chosen by choice operation for strategy variable x
$wasChosen[x]()$	some object was chosen for strategy variable x
$chosen(v)$	object v was chosen by some choice operation
$relevant(v)$	abstraction-directing predicate recording relevant objects

Table 2: Additional predicates of the instrumented semantics.

records the value of a boolean field `bf`. Finally, a unary predicate $site[AS](v)$ records the allocation site AS in which an object was allocated.

In order to enable interprocedural analysis we explicitly represent stack frames and a corresponding set of predicates following [16]. Since this does not interfere with the material in this paper, to simplify presentation we do not describe these predicates.

In this paper, program configurations are depicted as directed graphs. Each individual of the universe is drawn as a node. A unary predicate $p(o)$ which holds for a node u is drawn inside the node u. A binary predicate $p(u_1, u_2)$ which evaluates to 1 is drawn as directed edge from u_1 to u_2 labelled with the predicate symbol.

Example 1. Fig. 5 shows a concrete program configuration representing a global state of the program before executing the statement at line `28`. In this configuration, three `String` objects were allocated in the heap and are referenced by `maxQry`, `minQry`, `balancesQry`. The configuration also contains two `Connection` objects referenced by `con1` and `con2`, two `Statement` objects referenced by `stmt1` and `stmt2`, and three `ResultSet` objects referenced by `maxRs`, `rs1`, and `rs2`. Note that the `ResultSet` referenced by `maxRs` is closed. The meaning of the predicates $relevant(u)$, $chosen[c](u)$, $chosen[s](u)$, and $chosen[r](u)$ will become clear in the next section.

4.2 Instrumentation For Separation

In this section we explain how we translate a Java program, an `Easl` specification, and a strategy specification into a FOTS. Specifically, the strategy specification is used to instrument the standard translation of a Java program and `Easl` specification into a FOTS. (This translation also directly provides a formal semantics for a separation strategy as a method for non-deterministically choosing a set of objects during program execution.) We use the predicates in Table 2 to instrument the semantics. Predicates of the form $chosen[x](v)$, $wasChosen[x]()$, and $chosen(v)$ are used to express the separation strategy. The predicate $relevant(v)$ is an abstraction-directing predicate that controls the way in which an object is abstracted.

Consider a choice operation

```
choose all x :  T ($w_1$,...,$w_i$) / e($w_1$,...,$w_i$,$z_1$,...,$z_k$)
```

Here, we say that the choice operation binds variable `x`. Variables $\mathtt{w}_1$ through $\mathtt{w}_i$ are free variables corresponding to parameters of a call to a constructor for type `T`, while $\mathtt{z}_1$ through $\mathtt{z}_k$ are variables bound by earlier choice operations. In order to model the specified choice operation, we introduce an instrumentation predicate $chosen[\mathtt{x}](u)$. The idea is for the predicate $chosen[\mathtt{x}](u)$ to hold true for exactly the objects that are chosen by the above choice operation. We achieve this by translating the condition `e(...)` specified for the choice operation into a first-order logic formula which is evaluated on entry to the specified constructor `T` to compute the value of $chosen[\mathtt{x}](u)$ for the newly created object u. (Technically, this translation works by converting the free occurrences of a variable $\mathtt{z}_j$ by occurrences of an existentially quantified logical variable O_j that is constrained to satisfy predicate $chosen[\mathtt{z}_j](O_j)$.)

The translation of a `choose some x` operation is similar, except that the translation ensures that at most one of the objects that is eligible for selection by the operation is chosen. This is done by introducing a second instrumentation predicate $wasChosen[\mathtt{x}]()$ that indicates if an object has already been selected during program execution for the corresponding choice operation (thus, it is defined by the instrumentation formula $\exists O.chosen[\mathtt{x}](O)$). When a new `T` object O is constructed, $chosen[\mathtt{x}](O)$ is set to false if $wasChosen[\mathtt{x}]()$ evaluates to true or if the selection formula corresponding to the choice operation evaluates to false. Otherwise, $chosen[\mathtt{x}](O)$ is non-deterministically assigned either true or false, and $wasChosen[\mathtt{x}]()$ is correspondingly updated.

Given a simple strategy specification consisting of n choice operations over variables $\mathtt{z}_1$ through $\mathtt{z}_n$, we also introduce a unary predicate $chosen(O)$ that indicates if an object was chosen by any of the n choice operations: thus, it is defined by the instrumentation formula $chosen[\mathtt{z}_1](O) \vee \cdots \vee chosen[\mathtt{z}_n](O)$.

Finally, the actual checks on objects that verify they satisfy the necessary preconditions when methods are invoked on them are instrumented to do the check only for chosen objects.

For now, the predicate $relevant(u)$ may be thought of as being equivalent to $chosen(u)$. We will later see that the set of relevant objects includes all the chosen objects and potentially some other objects as well.

Example 2. The single-choice strategy for JDBC is modelled using predicates $chosen[c](u)$, $chosen[s](u)$, and $chosen[r](u)$. Upon entry to the constructor `Statement(Connection c)`, the condition of the corresponding choice operation is evaluated and the `Statement` is chosen if the passed `Connection` is the one for which $chosen[c](u)$ holds. Similarly, the condition for choosing a `ResultSet` is evaluated on entry to constructor `ResultSet(Statement s)`. As a result, for each subproblem $chosen[c](u)$ holds for (at most) a single `Connection` component, and $chosen[s](u)$, $chosen[r](u)$ hold for `Statement`s and `ResultSet`s that are related to the chosen `Connection`. Part of the instrumented program for this strategy is shown in Fig. 6 (For clarity, we use Easl syntax to present the instrumented program).

We now briefly indicate how incremental strategies are handled. The notion of a failed individual is fairly straightforward. A single strategy specification produces multiple verification subproblems, each over a set of chosen individuals. An individual is said to be a failed individual if it is a chosen individual of a verification subproblem that fails verification. However, we want to utilize simple strategy specifications that restrict their attention to individuals that failed the previous simple strategy specification. In general, this requires instrumentation that can identify at object-allocation time whether the allocated object corresponds to a failed individual in the previous verification step. This is hard to do in a very general way, and we restrict ourselves to allocation-site based identification of failed individuals: thus, if any one individual allocated at an allocation site fails verification, then all individuals allocated at that site are treated as failed individuals in the next verification step.

Operational Semantics

In a FOTS, program statements are modelled by *actions* that specify how the statement transforms an incoming logical structure into an outgoing logical structure. This is done primarily by defining the values of the predicates in the outgoing structure using first-order logic formulae with transitive closure over the incoming structure [17].

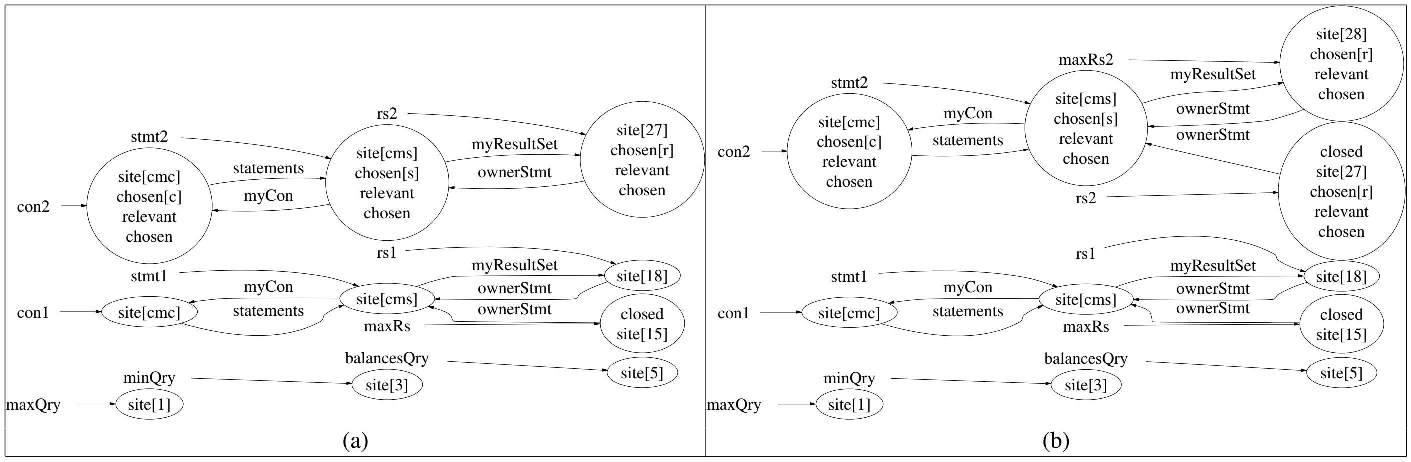

Figure 5: Concrete program configurations representing a possible program state (a) at line `28` and (b) after execution of statement at line `28`

```
class Connection {
  ...
  Connection() {
    if (!wasChosen) {
      if (?)  {
        chosen = true;
        wasChosen = true;
      } else
        chosen = false;
    }
    closed = false;
    statements = {};
  }
  Statement createStatement() {
    if (chosen)
      requires !closed;
    Statement st = new Statement(this);
    statements = statements U { st };
    return st;
  }
  ...
}
```

```
class Statement {
  ...
  Statement(Connection c) {
    chosen = c.chosen
    closed = false;
    myConnection = c;
    myResultSet = null;
  }
  ResultSet executeQuery(String qry) {
    if (chosen)
      requires !closed;
    if (myResultSet != null)
      myResultSet.closed = true;
    myResultSet = new ResultSet(this);
    return myResultSet;
  }
  ...
}
```

```
class ResultSet {
  ...
  ResultSet(Statement s) {
    chosen = s.chosen;
    closed = false ;
    ownerStmt = s;
  }
  ...
  boolean next() {
    if (chosen)
      requires !closed;
  }
}
```

Figure 6: An instrumented `Easl` specification for a simplified subset of the JDBC API with single-choice separation strategy.

Example 3. Fig. 5(b) shows the effect of the statement `maxRs2 = stmt2.executeQuery(maxQry)` at line `28`, where the statement is applied to the configuration in Fig. 5. The effect of the statement is reflected by its updates to predicate values. Here, we assume that the choice predicates and the instrumentation predicates are updated according to the single-choice strategy of Sec. 3. Since the constructor of the new `ResultSet` is invoked with a chosen `Statement` object, the choice condition is satisfied and the newly created `ResultSet` is chosen and made relevant.

4.3 Additional Instrumentation

The predicate *relevant* is intended to identify objects that must be modelled precisely for a verification subproblem. The separation strategy specification allows users to identify relevant objects (via choice clauses). An analysis designer, or a component library designer, can create separation strategies that reflect the dependencies that exist among component library objects, while an end user can create separation strategies that provide more dependency information (specific to their own program).

Currently, however, we do not assume that such extra dependency information will be available from an end user. Instead, we rely on a more automatic approach that considers objects which reach a relevant object as relevant themselves, thus creating a notion of *transitive relevance*. Transitive relevance causes all objects that are on a path to a relevant object to become relevant as well, thus separating heap paths that may reach a relevant object from heap paths that cannot.

We achieve this by defining the instrumentation predicate $relevant(u)$ to be true iff there is a path from u to some chosen object v (i.e., some object v for which $chosen(v)$ is true). We update this predicate using the techniques of [15].

5. HETEROGENEOUS ABSTRACTION

The essence of our separation-based verification is the following: first, a separation strategy is used to choose a set of objects (for a given program trace); second, we utilize specialized abstractions to perform verification for the chosen objects efficiently. These specialized abstractions represent the chosen objects much more precisely than the remaining objects. We refer to these abstractions as *heterogeneous* abstractions as they represent different parts of the heap with different degrees of precision. In this section we describe the abstractions we use for separation-based verification.

Abstract Program Configurations

The goal of an abstraction is to create a finite representation of a potentially unbounded set of 2-valued structures (representing heaps) of potentially unbounded size. The abstractions we use are based on 3-valued logic [17], which extends boolean logic by introducing a third value $1/2$ denoting values that may be 0 or 1.

Definition 2. A 3-valued logical structure over a set of predicates P is a pair $C = \langle U, \iota \rangle$ where:

- U is the universe of the 3-valued structure. An individual in U may represent multiple heap-allocated objects.
- ι is the interpretation function mapping predicates to their truth-value in the structure: for every predicate $p \in P$ of arity k, $\iota(p) : U^k \rightarrow \{ 0, 1, 1/2 \}$.

An abstract configuration may include *summary nodes*, i.e., an individual which corresponds to one or more individuals in a concrete configuration represented by that abstract configuration. We use a designated unary predicate *sm* to maintain summary-node information. A summary node u has $sm(u) = 1/2$, indicating that it may represent more than a single individual.

As in [17], the abstract interpretations we use work by abstracting the set of 2-valued structures that can arise at a program point by a set of 3-valued structures. However, this can be done in a number of ways as shown below.

Individual Merging. The basic abstraction primitive used by [17] is that of *individual merging*: a larger structure s can be safely approximated by a smaller 3-valued structure by merging multiple individuals into one, and by approximating the predicate values appropriately. Given an equivalence relation $\equiv$ on individuals, let $s/\equiv$ denote the structure obtained by merging individuals of s that are $\equiv$-equivalent together.

The above primitive induces a function $abs_1[\equiv]$ that abstracts a set of 2-valued structures by a set of 3-valued structures, defined by $abs_1[\equiv](S) = \{s/\equiv \mid s \in S\}$. (Strictly speaking, $abs_1[\equiv](S)$ retains only a single representative of isomorphic structures, but we ignore the fine distinction between isomorphism and equality for the sake of simplicity.)

[17] utilizes the equivalence relation $\equiv_A$ induced by a set of unary predicates A (referred to as the *abstraction* predicates) defined as follows: $o_1 \equiv_A o_2$ iff $p(o_1) = p(o_2)$ for every $p \in A$.

Structure Merging. Subsequently, TVLA [11] introduced more aggressive abstraction mechanisms based on the idea of *merging multiple structures* into one. Define the *union* $s_1 \cup s_2$ of two structures to be the structure whose universe is the disjoint union of the universes of s_1 and s_2, with the predicate interpretations of s_1 and s_2 extended appropriately. The union of a set of structures S is defined similarly. Structures are merged by first taking their union, and then merging individuals of the union along the lines indicated previously: define $\bigsqcup_{\equiv}(S)$ to be $(\bigcup S)/\equiv$.

Now, consider an equivalence relation $\simeq$ defined on *structures*, indicating which structures must be merged together, and an equivalence relation $\equiv$ defined on *individuals*. We can now define a parameterized abstraction function $abs_2[\simeq, \equiv](S)$ that first applies *individual merging* to every structure s in S, and then merges together the resulting structures that are $\simeq$-equivalent. Formally, $abs_2[\simeq, \equiv](S)$ is defined to be:

$$\{ \bigsqcup_{\equiv}(C) \mid C \text{ is an } \simeq\text{-equivalence class of } abs_1[\equiv](S) \}$$

TVLA utilizes the following $\simeq$ definitions: (a) $s_1 \simeq s_2$ iff s_1 and s_2 are isomorphic, (b) $s_1 \simeq s_2$ iff s_1 and s_2 have the same values for a specified set B of *nullary abstraction* predicates, (c) $s_1 \simeq s_2$ iff s_1 and s_2 have the same universes (modulo $\equiv$).

TVLA utilizes an extra unary predicate *active*, which indicates if an individual definitely exists in the universe or not, so that the structure $\bigsqcup_{\equiv}(S)$ can be used as an abstraction of every structure in S. Thus, if S is a set of 2-valued structures, then the predicate *active* is true for an individual o in $\bigsqcup_{\equiv}(S)$ iff the equivalence class represented by o includes at least one individual from every structure in S.

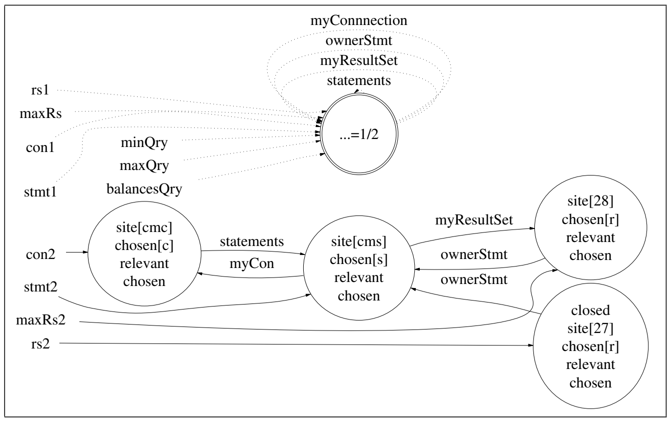

Figure 7: An abstract program configuration representing the concrete configuration of Fig. 5(b).

Heterogeneous Abstraction

Separation creates the possibility for achieving better efficiency by adapting the abstractions to model chosen individuals more precisely and the other individuals less precisely. In particular, this can be done by:

- *Adapting individual merging*: We can make finer distinctions between chosen individuals than for unchosen individuals, when we decide which individuals should be merged together. E.g., we can choose to use the less expensive allocation-site based merging for unchosen individuals, and more expensive variable-name based merging for chosen individuals.
- *Adapting structure merging*: Similarly, when deciding which structures should be merged into one, we could choose to treat chosen and unchosen individuals differently.
- *Adapting predicate values retained*: One could choose to not record the values of certain predicates for unchosen individuals. While this can reduce the space required to represent a structure, this does not, unlike the preceding techniques, reduce the number of structures in the abstraction. We will not discuss this in this paper.

We now define a new family of equivalence relations for identifying individuals to be merged. Consider a quadruple $\langle c, A_1, A_0, A_{1/2} \rangle$ where c is a unary predicate, and A_1, A_0, and $A_{1/2}$ are all sets of unary predicates. The equivalence relation $\equiv_{\langle c, A_1, A_0, A_{1/2} \rangle}$ on individuals is defined by:

$$\begin{gathered}(c(o_1) = c(o_2) = 1) \wedge \forall p \in A_1 . p(o_1) = p(o_2)) \vee \\ ((c(o_1) = c(o_2) = 0) \wedge \forall p \in A_0 . p(o_1) = p(o_2)) \vee \\ ((c(o_1) = c(o_2) = 1/2) \wedge \forall p \in A_{1/2} . p(o_1) = p(o_2))\end{gathered}$$

Given a set Γ of such tuples, we define $\equiv_\Gamma$ to be $\bigsqcap_{\gamma \in \Gamma} \equiv_\gamma$.

We similarly define a new criteria for structure merging. Given a unary predicate c, define $s_1 \simeq_c s_2$ iff the substructures of s_1 and s_2 consisting only of individuals i for which $c(i) = 1$ are isomorphic.

For our separation-based verification, we utilize the abstraction induced by the equivalence relations $\equiv_{\langle relevant, A, \emptyset, A\rangle}$ and $\simeq_{relevant}$, where A is the set of abstraction predicates utilized by the underlying separation-less verification. (In our implementation, this consists of the set of unary predicates).

Implementation Notes. Our current implementation uses a very close approximation of the individual merging induced by the equivalence relation $\equiv_{\langle relevant, A, \emptyset, A\rangle}$ as follows: for every predicate p in A, we introduce a new instrumentation predicate $p_r(o) = p(o) \wedge relevant(o)$, and use the set of predicates $\{ \ p_r \mid p \in A \ \}$ as the set of abstraction predicates.

Example 4. Fig. 7 shows an abstract configuration representing the concrete configuration of Fig. 5(b), obtained by heterogeneous relevance-based abstraction. Abstract program configurations are depicted similarly to concrete configurations with an additional representation of summary nodes as nodes with double-line boundaries, and a $1/2$-valued binary predicate as a dashed edge. All individuals for which $relevant$ holds are abstracted by the values of the predicates in A_1. Other individuals, for which $relevant$ does not hold, are merged into a single summary node since $A_0 = \emptyset$. In particular, this abstract configuration abstracts away the current state of objects related to `Connection con1`, including the state of `Statement stmt1`. In the figure, we use $\ldots = 1/2$ instead of listing all predicates that have $1/2$ value for the summary node.

If we had used a "homogeneous" abstraction, the non-relevant objects would have been abstracted using the same set of predicates as the relevant objects (A_1), thus keeping the objects related to the `Connection` referenced by `con1` with the same precision, and cost, as the ones related to `Connection` referenced by `con2`. The ability to treat these structurally-similar objects very differently during analysis is a key to obtaining good results with our method.

Abstract Semantics

We will now briefly describe the abstract semantics ("transfer functions") we utilize for program statements.

A key idea underlying [17] is that the actions defining a standard operational semantics for a program statement (as a transformer of 2-valued structures) also define a corresponding abstract semantics for the statement (as a transformer of 3-valued structures). This abstract semantics is simply obtained by reinterpreting logical formulae using a 3-valued logic semantics and serves as the basis for an abstract interpretation. However, [17] also presents techniques, such as materialization, that improve the precision of such an abstract semantics. We directly utilize the implementation of these ideas available in TVLA.

We described earlier (see Sec. 4.2) how we utilize instrumentation predicates to identify relevant objects. We currently also utilize instrumentation predicates to achieve a heterogeneous abstraction. We use the techniques in [15] for automatically generating, from the instrumentation formula, an instrumented abstract semantics for statements to update the values of these instrumentation predicates.

6. PROTOTYPE IMPLEMENTATION

We have implemented a prototype of the separation verification engine using TVLA [11]. To translate Java programs and their specifications to TVP (TVLA input language) we have extended an existing Soot-based [20] front-end for Java developed by R. Manevich.

The implementation emulates heterogeneous abstraction using instrumentation predicates in TVLA, which adds some overhead. We believe that a native implementation of heterogeneous abstraction will yield better performance.

Program	Description	Mode	Line No.	Space (MB)	Time (Sec)	Rep. Err.	Act. Err.
ISPath	inp. streams	vanilla	71	9.17	145.5	0	0
	/ IOStreams	single		2.51	17.4	0	
		sim		3.94	12.3	0	
Input	inp. streams	vanilla	64	16.35	439	1	0
Stream5	holders	single		17.65	240	0	
	/ IOStreams	sim		21.35	202	0	
Input	inp. streams	vanilla	64	13.72	343	1	1
Stream5b	holders err	single		19.71	279	1	
	/IOStreams	sim		22.74	243	1	
Input	inp. streams	vanilla	66	37.17	1344	1	0
Stream6	holders	single		13.91	69.4	1	
	/ IOStreams	sim		12.14	51.3	1	
JDBC	extended	vanilla	149	33.43	2500	1	1
Example	example	single		28.71	1090	1	
	/ JDBC	multi		16	7340	1	
		inc		12.5	3579	1	
JDBC	extended	vanilla	153	32.8	2500	0	0
Example	example	single		28.8	1090	0	
fixed	/ JDBC	multi		29.5	7500	0	
		inc		25.7	3339	0	
db	SpecJVM98	vanilla	644	89.25	10454	0	0
	db	single		90	2500	0	
	/ IOStreams	sim		91.17	1496	0	
Kernel	Collections	vanilla	82	42.23	8321	1	1
Bench.1	benchmark	single		13.15	657	1	
	/ CMP	sim		13.84	255	1	
		multi		14.45	4552	1	
		inc		14.45	960	1	
Kernel	Collections	vanilla	146	–	–	–	1
Bench.3	benchmark	single		107.8	12098	1	
	/ CMP	sim		128.7	7588	1	
		multi		119	69631	1	
		inc		106	12881	1	
SQL	JDBC	vanilla	1297	–	–	–	0
Executor	framework	single		80.59	5028	0	
	/ JDBC	multi		72.64	4919	0	
		inc		42.68	412	0	

Table 3: Analysis results and cost for the benchmark programs.

We applied our framework to verify various specifications for a number of example programs. Our specifications include correct usage of JDBC, IO streams, Java collections and iterators, and additional small but interesting specifications. The experiments were performed on a machine with a 1 Ghz Pentium 4 processor, 1 Gb[2]. Results are shown in Table 3. The column titled "mode" shows the analysis mode for each line in the table. Verification with TVLA with no separation is referred to as *vanilla* mode. "Rep. Err." shows the number of reported errors, while "Act. Err." shows the number of actual errors. When counting errors, we count all errors reported at the same program location as a single error.

Our implementation allows control over which subproblems are verified simultaneously. This allows verification of subproblems related to one (or more) allocation-sites separately from other subproblems, reducing the maximal memory footprint of the verification. The measurements in Table 3 correspond to this non-simultaneous mode. The space measurement shown in Table 3 for separation modes (*single*, *multi*, *incremental*) is the maximal space required for analyzing a single set of subproblems. The time is the accumulated time for analyzing all subproblems. The table also shows measurements for simultaneous verification of all subproblems using single-mode (*sim* mode). For the JDBC example, the simultaneous single-choice mode is identical to the non-simultaneous mode.

[2]SQLExecutor analyzed on a machine with a 2.79Ghz processor.

ISPath is a simple correct program manipulating input streams. InputStream5 is a heapful example program that manipulates inputstreams in holder objects at arbitrary depth of the heap. For this program, the vanilla version produces a false-alarm that is avoided by the separation-based analysis. This is due to the use of *transitive relevance* which makes the separation-based analysis more precise (for the relevant objects). Generally, since the separation-based analysis is more focused, it may allow using a more precise abstraction than the one that could be used when applied uniformly. InputStream5b is an erroneous version of InputStream5 containing a single error. InputStream6 is another variation of InputStream5.

JDBCExample is an extended version of the running example that uses 5 `Connections`. The high running-time result for incremental mode in this case is affected by the fact that there is small number of `Statements` (1) and `ResultSets` (up to 3) associated with each `Connection`. db is a program from SpecJVM98 performing multiple database functions on a memory resident database.

KernelBenchmark1 and KernelBenchmark3 are part of a benchmark suite for testing Collections and Iterators used in [14]. SQLExecutor is an open source JDBC framework. For this benchmark, vanilla verification failed to terminate after more than 5 hours, but incremental-mode successfully verified the program in 412 seconds. This is a result of the correct and relatively simple usage of JDBC objects in this benchmark.

In some benchmarks separation gained an overall performance increase, while in others the total verification time in some modes was larger than the time for vanilla-mode verification. In all cases, however, the average time for verifying a single subproblem was significantly lower than the time required for vanilla verification. Thus, separation may be useful for answering on-demand queries when one is only interested in checking whether an object (or a set of correlated objects) can produce an error. E.g., while the total time for multi-mode and incremental-mode in the JDBC example was larger than the time required for vanilla-mode, the average time for verifying each subproblem was approximately 670 seconds.

One interesting future direction is to exploit separation for increasing performance by parallelizing verification of subproblems.

7. EXTENSIONS AND FUTURE WORK

We have experimented with two classes of iterative refinement schemes for approximating the set of relevant objects for a subproblem: the first iteratively identifies more "relevant program variables" and turns objects pointed-to by these variables relevant; the second iteratively identifies "relevant allocation sites" and turns objects allocated at these sites relevant. Both classes of our refinement schemes are guaranteed to terminate (with all objects being relevant in the worst case), but are not guaranteed to yield a successful verification. Our initial experience indicates that these techniques work well for relatively small examples.

8. ACKNOWLEDGEMENTS

The authors wish to thank John Field and Mooly Sagiv for their insightful comments, which helped improve the paper, and Roman Manevich for his considerable contribution to the implementation.

9. REFERENCES

[1] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *Proc. IEEE Symp. on Security and Privacy*, Oakland, CA, May 2002.

[2] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001*, LNCS 2057, pages 103–122, 2001.

[3] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. Intl. Conf. on Software Eng.*, pages 439–448, June 2000.

[4] J. Corbett, M. Dwyer, J. Hatcliff, and Robby. Expressing checkable properties of dynamic systems: the bandera specification language. *STTT*, 4(1):34–56, Oct. 2002.

[5] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proc. Conf. on Prog. Lang. Design and Impl.*, pages 57–68, June 2002.

[6] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. Conf. on Prog. Lang. Design and Impl.*, pages 59–69, June 2001.

[7] J. Field, D. Goyal, G. Ramalingam, and E. Yahav. Typestate verification: Abstraction techniques and complexity results. In *Proc. of SAS'03*, volume 2694 of *LNCS*, pages 439–462. Springer, June 2003.

[8] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proc. Conf. on Prog. Lang. Design and Impl.*, pages 234–245, Berlin, June 2002.

[9] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proc. Conf. on Prog. Lang. Design and Impl.*, pages 1–12, Berlin, June 2002.

[10] S. Guyer and C. Lin. Client-driven pointer analysis. In *Proc. of SAS'03*, volume 2694 of *LNCS*, pages 214–236, June 2003.

[11] T. Lev-Ami and M. Sagiv. TVLA: A framework for Kleene based static analysis. In *Proc. Static Analysis Symp.*, volume 1824 of *LNCS*, pages 280–301. Springer-Verlag, 2000.

[12] K. L. McMillan. Verification of infinite state systems by compositional model checking. In *Proc. of CHARME '99*, volume 1703 of *LNCS*, pages 219–237, 1999.

[13] Microsoft Research. The SLAM project. http://research.microsoft.com/slam/, 2001.

[14] G. Ramalingam, A. Warshavsky, J. Field, D. Goyal, and M. Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *Proc. Conf. on Prog. Lang. Design and Impl.*, volume 37, 5, pages 83–94, June 2002.

[15] T. Reps, M. Sagiv, and A. Loginov. Finite differencing of logical formulas for static analysis. In *In Proc. European Symp. on Programming*, 2003.

[16] N. Rinetzky and M. Sagiv. Interprocedural shape analysis for recursive programs. *LNCS*, 2027:133–149, 2001.

[17] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. on Prog. Lang. and Systems (TOPLAS)*, 24(3):217–298, 2002.

[18] R. Shaham, E. Yahav, E. Kolodner, and M. Sagiv. Establishing local temporal heap safety properties with applications to compile-time memory management. In *Proc. of SAS'03*, volume 2694 of *LNCS*, pages 483–503, June 2003.

[19] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986.

[20] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a java optimization framework. In *Proc. of CASCON 1999*, pages 125–135, 1999.

[21] S. White, M. Fisher, R. Cattell, G. Hamilton, and M. Hapner. *JDBC API tutorial and reference*. Addison-Wesley, 1999.