TVLA: A SYSTEM FOR GENERATING ABSTRACT INTERPRETERS*

Tal Lev-Ami, Roman Manevich, and Mooly Sagiv
Tel Aviv University
{tla@trivnet.com, {rumster,msagiv}@post.tau.ac.il}

Abstract TVLA (Three-Valued-Logic Analyzer) is a "YACC"-like framework for automatically constructing abstract interpreters from an operational semantics. The operational semantics is specified as a generic transition system based on first-order logic. TVLA was implemented in Java and successfully used to prove interesting properties of (concurrent) Java programs manipulating dynamically allocated linked data structures.

1. Introduction

The abstract-interpretation technique of Cousot and Cousot, 1979 for static analysis allows one to summarize the behavior of a statement on an infinite set of possible program states. This is sometimes called an *abstract semantics* for the statement. With this methodology it is necessary to show that the abstract semantics is *conservative*, i.e., it summarizes the (*concrete*) *operational semantics* of the statement for every possible program state. Intuitively speaking, the operational semantics of a statement is a formal definition of an interpreter for this statement. This operational semantics is usually quite natural. However, designing and implementing sound and reasonably precise abstract semantics is quite cumbersome (the best induced abstract semantics defined in Cousot and Cousot, 1979 is usually not computable). This is particularly true in problems like shape analysis and pointer analysis (e.g., see Sagiv et al., 1998; Sagiv et al., 2002), where the operational semantics involves destructive memory updates.

*Supported, in part, by a grant from the Academy of Science, Israel.

1.1 An Overview of the TVLA System

In this paper, we review TVLA (**T**hree-**V**alued-**L**ogic **A**nalyzer), a system for automatically generating a static-analysis implementation from the operational semantics of a given program (Lev-Ami and Sagiv, 2000). The small-step structural operational semantics is written in a meta-language based on first-order predicate logic with transitive closure. The main idea is that program states are represented as logical structures and the program transition system is defined using first-order logical formulas. TVLA automatically generates the abstract semantics, and, for each program point, produces a conservative abstract representation of the program states at that point. The idea of automatically generating abstract semantics from concrete semantics was proposed in Cousot, 1997.

TVLA is intended as a proof of concept for abstract interpreters. It is a test-bed in which it is quite easy to try out new ideas. The theory behind TVLA is based on Sagiv et al., 2002.

Static Analysis Using TVLA A front-end `J2TVLA` converts a Java program into `tvp`, the input meta-language of TVLA. This front-end is available separately and is not further described in this document. A typical input of TVLA consists of four text files: (i) The type of concrete states of the analyzed programs is defined in the file `pred.tvp`. This file defines predicates (relation symbols) which hold concrete values of variables and program stores. (ii) The meaning of atomic program statements and conditions is defined in the action file `acts.tvp`. Actions allow to naturally model program conditions and mutations of stores. They are defined using first-order logical formulas. TVLA actions can also produce error messages when safety violations occur. Both actions and predicates are usually defined once for a given analysis. They are parameterized by information specific to the analyzed program, such as the names of program variables, types, fields, and classes. (iii) A file `fots.tvp` defines the transition system of the analyzed program. It is basically a control flow graph with edges annotated by actions from the action file, and can be automatically generated by `J2TVLA` for Java programs. (iv) The `tvs` file `init.tvs` describes the abstract value at the program entry. It can be used to allow modular TVLA analysis of a separate program component, which does not start with an empty store.

The core of the TVLA engine is a standard chaotic iteration procedure, where actions are conservatively interpreted over an abstract domain of 3-*valued structures.* This means that the system guarantees that no safety violation is missed but it may produce ”false alarms”, i.e., warnings about violations that can never occur in any concrete execution. Finally, TVLA allows to investigate the output 3-valued structures, which can either be displayed in `Postscript` format or as a `tvs` file `out.tvs` to be read by other tools.

The unique part of TVLA is the automatic generation of the abstract interpretation of actions in a way that is: (i) guaranteed to be sound, and (ii) rather precise—the number of false alarms in our applications is very small.

Outline The rest of this tutorial is organized as follows: In Sect. 2 we describe the TVLA meta-language for constructing concrete semantics; In Sect. 3 we provide an overview of 3-valued logical based static analysis; In Sect. 4 we describe several enhancements and applications of the system; and in Sect. 5 we give concluding remarks.

2. First-Order Transition Systems

We now present an overview of *first order transition systems* (FOTS). In FOTS, program statements are modelled by *actions* that specify how the statement transforms an incoming logical structure into an outgoing logical structure.

A Running Example Fig. 2 shows our running example—a method implementing the Mark phase of a mark-and-sweep garbage collector and its transition system. The challenge here is to show that this method is partially correct, i.e., to establish that "upon termination, an element is marked if and only if it is reachable from the root." TVLA successfully verifies this correctness property in 5 CPU seconds.

2.1 Concrete Program States

In FOTS, program states are represented using 2-valued logical structures.

In the context of heap analysis, a logical structure represents the memory state (heap) of a program, with each individual corresponding to a heap-allocated object and predicates of the structure corresponding to properties of heap-allocated objects.

Table 1 shows the predicates we use to record properties of individuals for the analysis of our running example. 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) field `fld`; in our example `fld` $\in$ {`left`, `right`}. A unary predicate $set[s](v)$ holds when the object v belongs to the set s;

```
//@Ensures marked == REACH(root)
void mark(Node root, NodeSet marked){
  Node x, t;
  if (root != null) {
     NodeSet pending = new NodeSet();
     pending.add(root);
     marked = new NodeSet();
     while (!pending.isEmpty()) {
       x = pending.selectAndRemove();
       marked.add(x);
       t = x.left;
       if (t != null)
          if (!marked.contains(t))
             pending.add(t);
          x = x.right;
          if (t != null)
             if (!marked.contains(t)
                pending.add(t);
       }
     }
  }
```

```
n0  IsNotNull(root)              n1
n1  AssignEmpty(pending)         n2
n2  Add(pending,root)            n3
n3  AssignEmpty(marked)          n4
n4  NotIsEmpty(pending)          n5
n4  IsEmpty(pending)             n17
n5  SelectAndRemove(pending,x)   n6
n6  Add(marked,x)                n7
n7  Load(t,x,left)               n8
n8  IsNotNull(t)                 n9
n8  IsNull(t)                    n12
n9  NotContains(marked,t)        n11
n9  Contains(marked,t)           n12
n11 Add(pending,t)               n12
n12 Load(t,x,right)              n13
n13 IsNotNull(t)                 n14
n13 IsNull(t)                    n4
n14 NotContains(marked,t)        n16
n14 Contains(marked,t)           n4
n16 Add(pending,t)               n4
n17 NotEqualReach(marked,root)   error
n17 EqualReach(marked,root)      exit
```

Figure 1. A simple Java-like implementation of the mark phase of a mark-and-sweep garbage collector and its transition system.

Table 1. Predicates used to verify the running example.

Predicates	**Intended Meaning**
$x(v)$	reference variable x points to the object v
$t(v)$	reference variable t points to the object v
$root(v)$	reference variable *root* points to the object v
$\texttt{left}(v_1, v_2)$	field `left` of the object v_1 points to the object v_2
$\texttt{right}(v_1, v_2)$	field `right` of the object v_1 points to the object v_2
$set[marked](v)$	object v is a member of the *marked* set
$set[pending](v)$	object v is a member of the *pending* set
$r[root](v)$	object v is heap-reachable from reference variable *root*

in our example $s \in \{marked, pending\}$. The predicate $r[root](v)$ is a special kind of predicate, used to record reachability information. It is not needed to define the concrete semantics, but is needed to refine the abstraction. Here, it is used to distinguish between individuals that are reachable from the root variable and individuals that are garbage. Predicates of this kind are called "instrumentation predicates".

In this paper, program states (i.e., 2-valued logical structures) 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

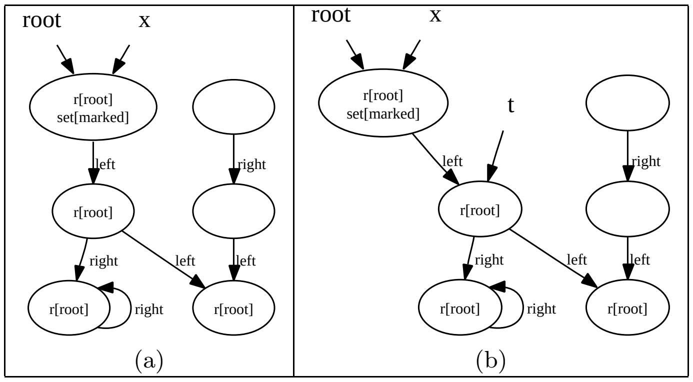

Figure 2. (a) A concrete program state arising before the statement `t = x.left`; (b) A concrete program state arising after the statement `t = x.left`.

inside the node u. If a unary predicate represents a reference variable it is shown by having an arrow drawn from its name to the node pointed by the variable. A binary predicate $p(u_1, u_2)$ that evaluates to 1 is drawn as a directed edge from u_1 to u_2 labelled with the predicate symbol.

Fig. 2(a) shows an example of a concrete program state arising before the statement `t = x.left`.

3. 3-Valued-Logic-Based Analysis

We now describe the abstraction used to create a finite (bounded) representation of a potentially unbounded set of 2-valued structures of potentially unbounded size. The abstraction is based on 3-valued logic, which extends boolean logic by introducing a third value 1/2 denoting values that may be 0 or 1.

A 3-valued logical structure can be used as an abstraction of a larger 2-valued logical structure. This is achieved by letting an abstract state (i.e., a 3-valued logical structure) to include *summary nodes*, i.e., individuals that correspond to one or more individuals in a concrete state represented by that abstract state. During the sequel of this paper, we will assume that the set of predicates P includes a distinguished unary predicate sm to indicate if an individual is a summary node.

In this paper, 3-valued logical structures are also depicted as directed graphs, where binary predicates with 1/2 values are shown as dotted edges and summary individuals are shown as double-circled nodes.

TVLA relies on a fundamental abstraction operation for converting a potentially unbounded structure into a bounded 3-valued structure. This abstraction operation is parameterized by a special set of unary predicates A referred to as the *abstraction* predicates.

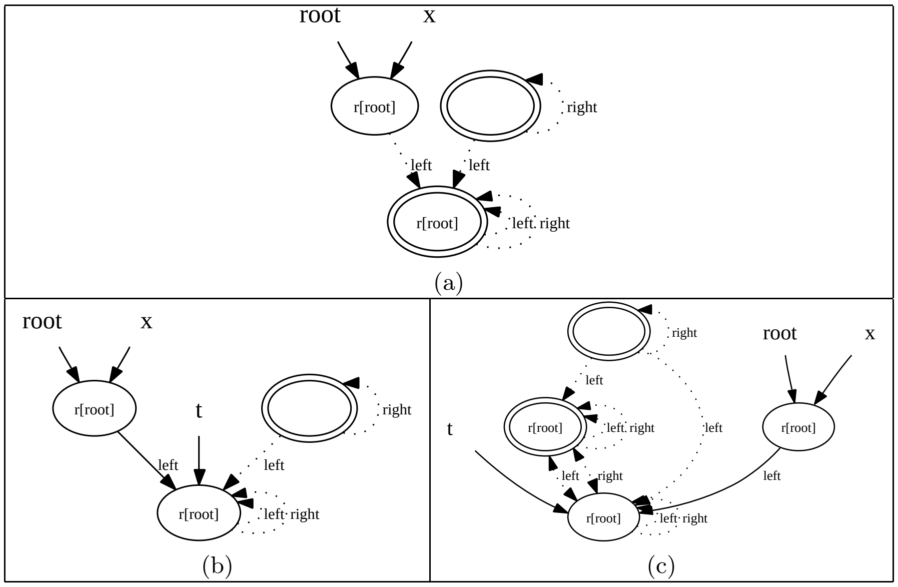

Figure 3. (a) An abstract program state approximating the concrete program state shown in Fig. 2(a); (b) and (c) are the abstract program states resulting from the abstract interpretation of the action `Load(t,x,left)`.

Let A be a set of unary predicates. Individuals u_1 and u_2 in a structure S are said to be A-equivalent iff for every predicate $p \in A$, $p^S(u_1) = p^S(u_2)$. A 3-valued structure is said to be A-bounded if no two different individuals in its universe are A-equivalent.

Informally, an A-bounded structure can be obtained from any structure by merging all pairs of A-compatible nodes, resulting in a structure with at most $2^{|A|}$ individuals that approximates the original (non-bounded) structure.

Fig. 3(a) shows an A-bounded structure obtained from the structure in Fig. 2(a) with $A = \{x, t, root, r[root], set[marked], set[pending]\}$.

3.1 Abstract Semantics

TVLA automatically produces an implementation of an abstract transformer for every action, which is guaranteed to be a conservative approximation of that action. Users can tune the transformer to achieve a high degree of precision. For example, in Fig. 3, the application of the transformer of the action `Load(t,x,left)` to the structure in (a) results in the structures shown in (b) and (c). In this case, the result is identical to the result of the best (most precise) transformer.

4. TVLA Enhancements and Applications

In this section we sketch several TVLA enhancements that were implemented in order to increase applicability, and mention some applications.

Algorithm Explanation by Shape Analysis In Bieber, 2001, TVLA is extended with visualization capabilities to allow re-playing changes in abstract states along different control-flow paths.

Finite Differencing In Reps et al., 2003, an algorithm for generating predicate-update formulas for instrumentation predicates is described. This technique is applied to generate predicate-update formulas for intricate procedures manipulating tree data structures.

Automatic Generation of Instrumentation Predicates In Ramalingam et al., 2002, a technique for generating instrumentation predicates based on backward weakest preconditions is described. This technique is applied to verify the absence of concurrent modification exceptions in Java. In Loginov et al., 2004, orthogonal techniques for (forward) generation of instrumentation predicates are applied to prove the correctness and stability of sorting algorithms.

Compactly Representing 3-Valued Structures In Manevich et al., 2002, the space cost of TVLA is reduced by representing 3-valued structures with data structures that share equivalent sub-parts.

Partially Disjunctive Abstractions In Manevich et al., 2004, the cost of TVLA analyses is reduced by applying more aggressive abstractions. The running time of the running example is reduced from 579 CPU seconds to 5 CPU seconds.

Numeric Abstractions In Gopan et al., 2004 it is shown how to handle numeric properties for an unbounded number of elements. This allows more precise and more automatic analyses using existing numeric abstractions. The method is applied to show absence of array bound violations in a program implementing sparse matrix multiplications using double indirections (i.e., `a[b[j]]`).

Best Transformers In Yorsh et al., 2004, theorem-provers are harnessed to compute the best (induced) transformers for 3-valued structures. This can be applied for modular assume-guarantee abstract interpretation in order to handle large programs with partial specifications.

Heterogenous Abstractions In Yahav and Ramalingam, 2004, a framework for *heterogeneous abstraction* is proposed, allowing different parts of the heap to be abstracted with different degrees of precision at different points during the analysis. The framework is applied to prove correct usage of JDBC objects and I/O streams, and absence of concurrent modifications in Java collections and iterators.

Interprocedural Analysis In Rinetzky and Sagiv, 2001, TVLA is applied to handle procedures by explicitly representing activation records as a linked list, allowing rather precise analysis of recursive procedures.

Concurrent Java Programs Yahav, 2001 presents a general framework for proving safety properties of concurrent Java programs with unbounded number of objects and threads. In Yahav and Sagiv, 2003 it is applied to verify partial correctness of a two-lock queue implementation.

Temporal Properties Yahav et al., 2003 proposes a general framework for proving temporal properties by representing program traces as logical structures. A more efficient technique for proving local temporal properties is presented in Shaham et al., 2003 and applied for compile-time garbage collection in Javacard programs.

5. Conclusion

TVLA is a system for generating implementations of static analysis algorithms, successfully used for a wide range of applications. Several aspects contributed to the usefulness of the system:

Firm theoretical background TVLA is based on the theoretical framework of Sagiv et al., 2002, which provides a proof of soundness via the embedding theorem. This relieves users from having to prove the soundness of the analysis.

Powerful meta-language The language of first-order logic with transitive closure is highly expressive. Users can specify different verification properties, and model semantics of different programming languages and different programming paradigms.

Automation and flexibility TVLA generates several ingredients that are essential for a precise static analysis. Users can tune the precision and control the cost of the generated algorithm.

Although TVLA is useful for solving different problems, it has certain limitations. The cost of the generated algorithm can be quite prohibitive, preventing analysis of large programs. Some of the costs can be reduced by better engineering certain components and other costs can be reduced by developing more efficient abstract transformers. The problem of generating more precise algorithms deserves further research.

References

Bieber, R. (2001). Alexsa—algorithm explanation by shape analysis—extensions to the TVLA system. Diplomarbeit, Universität des Saarlandes, Saarbrücken, Germany.

Cousot, P. (1997). Abstract interpretation based static analysis parameterized by semantics. In *Static Analysis Symp.*, pages 388–394.

Cousot, P. and Cousot, R. (1979). Systematic design of program analysis frameworks. In *Symp. on Princ. of Prog. Lang.*, pages 269–282, New York, NY. ACM Press.

Gopan, D., DiMaio, F., Dor, N., Reps, T., and Sagiv, M. (2004). Numeric domains with summarized dimensions. In *Int. Conf. on Tools and Algs. for the Construction and Analysis of Systems*, pages 512–529.

Lev-Ami, T. and Sagiv, M. (2000). TVLA: A system for implementing static analyses. In *Static Analysis Symp.*, pages 280–301.

Loginov, A., Reps, T., and Sagiv, M. (2004). Abstraction refinement for 3-valued-logic analysis. Tech. Rep. 1504, Comp. Sci. Dept., Univ. of Wisconsin.

Manevich, R., Ramalingam, G., Field, J., Goyal, D., and Sagiv, M. (2002). Compactly representing first-order structures for static analysis. In *Static Analysis Symp.*, pages 196–212.

Manevich, R., Sagiv, M., G.Ramalingam, and J.Field (2004). Partially disjunctive heap abstraction. In *Static Analysis Symp.* To appear.

Ramalingam, G., Warshavsky, A., Field, J., Goyal, D., and Sagiv, M. (2002). Deriving specialized program analyses for certifying component-client conformance. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 83–94.

Reps, T., Sagiv, M., and Loginov, A. (2003). Finite differencing of logical formulas for static analysis. In *European Symp. on Programming*, pages 380–398.

Rinetzky, N. and Sagiv, M. (2001). Interprocedural shape analysis for recursive programs. In Wilhelm, R., editor, *Proc. of CC 2001*, volume 2027 of *LNCS*, pages 133–149. Springer.

Sagiv, M., Reps, T., and Wilhelm, R. (1998). Solving shape-analysis problems in languages with destructive updating. *Trans. on Prog. Lang. and Syst.*, 20(1):1–50.

Sagiv, M., Reps, T., and Wilhelm, R. (2002). Parametric shape analysis via 3-valued logic. *Trans. on Prog. Lang. and Syst.*

Shaham, R., Yahav, E., Kolodner, E., and Sagiv, M. (2003). 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.

Yahav, E. (2001). Verifying safety properties of concurrent Java programs using 3-valued logic. In *Symp. on Princ. of Prog. Lang.*, pages 27–40.

Yahav, E. and Ramalingam, G. (2004). Verifying safety properties using separation and heterogeneous abstractions. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.* To appear.

Yahav, E., Reps, T., Sagiv, M., and Wilhelm, R. (2003). Verifying temporal heap properties specified via evolution logic. In *European Symp. on Programming*, volume 2618 of *LNCS*.

Yahav, E. and Sagiv, M. (2003). Automatically verifying concurrent queue algorithms. In *Workshop on Software Model Checking.*

Yorsh, G., Reps, T., and Sagiv, M. (2004). Symbolically computing most-precise abstract operations for shape analysis. In *Int. Conf. on Tools and Algs. for the Construction and Analysis of Systems*, pages 530–545.